École des Ponts ParisTech

Master II – End-of-studies intership report

Development of a finite volume library on GPUs using the adaptive mesh refinement library t8code

Author:
Maël Karembe

Supervisors:
Johannes Markert
Johannes Holke

Contents

1 Introduction

1.1 Context

In recent years, supercomputers started relying more heavily on GPUs due to their power efficiency and massively parallel capabilities in order to accomodate machine learning workloads and reach the exascale. For instance, the record breaking Frontier supercomputer contains 9,472 AMD Epyc 7713 Trento CPUs, each having 64 cores and 37,888 Instinct MI250X GPUs for a total of 8,335,360 cores. On this machine, each node contains one CPU for 4 GPUs. Therefore, for numerical simulation, it is necessary to develop software to target those new hybrid architectures to reach exascale computing. Those new heterogeneous machines introduce numerous challenges as operation on a GPU are fundamentally different from conventional CPUs. Thus, programming for GPU is a very demanding task requiring handling communication between CPU and GPU, managing two different address spaces, generating GPU-specific code. In addition, multi-node multi-CPU/GPU adds another layer of complexity with the necessity of coupling multiple parallel programming paradigms (MPI and CUDA for example). A lot of hardware details need to be known in order to efficiently use those machines. Therefore, building abstraction layers imitating GPU architecture are needed to increase programmer productivity while ensuring maximum performance and portability.

Furthermore, to more efficiently use compute resources, dynamic mesh adaptation is a very effective technique to focus computational capabilities on regions of interest and is used in many scientific fields: astrophysics, fluid dynamics. However, this technique as it is, is not well suited for GPUs as complex dynamic logic flow is required. In addition, doing mesh adaptation incurs an extra overhead that needs to be offset by the gain in efficiency. Doing mesh adaptation entirely on the GPU is tricky as it is not a task that suits the GPU execution model and thus requires designing new algorithms. Doing refinement on the CPU and computation on the GPU also has difficulties as data transfers between the CPU and GPU need to be minimized as CPU/GPU communication is relatively slow. Nevertheless, both approaches have been explored in many CPU/GPU AMR frameworks/codes.

1.2 Objectives

The objective of this end-of-studies intership is to develop a prototype numerical simulation framework using the adaptive mesh refinement library t8code developed by the work group *scalable adaptive mesh refinement* where I did this internship at the DLR (Deutsches Zentrum für Luft- und Raumfahrt). This framework should expose an API that allows any user to easily implement a finite volume solver without compromising ease of use for performance. After having done that, the goal is to assess t8code's suitability as an AMR library used to do numerical simulation on GPUs while doing mesh adaptation on the CPU.

This report is structured in multiple sections: firstly, we will detail the difference between GPU and CPU architectures to show what are the challenges when programming on GPUs and present various GPU programming models and optimization considerations. Then, we will present the different ways of doing adaptive mesh refinement and focus on t8code's approach to AMR. Afterwards, we will present the framework that I developed as well as the design choices behind it. Next, we will show performance results and end with conclusions and perspectives by analysing the overhead of AMR and also the computational efficiency of the GPU solver.

2 GPGPU programming concepts

Before delving into the GPU architecture, it is a good idea to summarize quickly the modern CPU architecture to see what sets appart GPU from CPU architecture and how targetting and optimizing for such devices require a different model.

2.1 CPU architecture strenghs and weaknesses

A modern CPU is composed of multiple cores, each core being able to run one (or more with hyperthreading or simultaneous multithreading SMT) execution thread at a time. Each core of a CPU is optimized to reduce **latency** *i.e.* minimize the number of cycles each CPU instruction takes to be executed. Multiple optimizations are taken to drastically reduce execution time. The most important ones are:

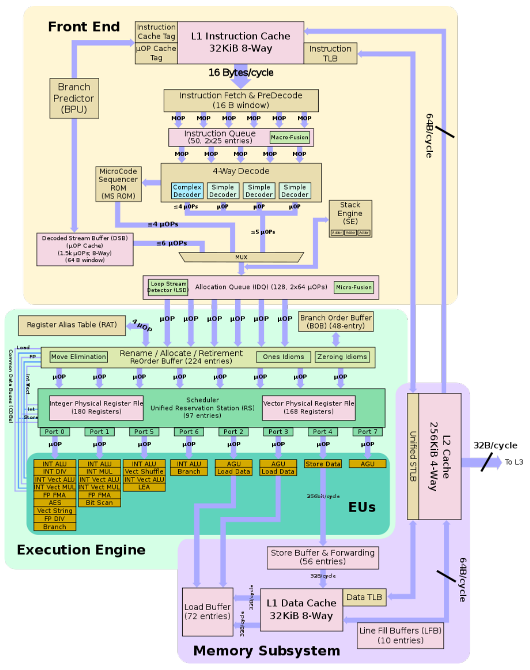

Front End
L1 Instruction Cache 32KiB 8-Way
Instruction Cache Tag
µOP Cache Tag
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
Instruction Queue (50, 2x25 entries)
Macro-Fusion
MicroCode Sequencer ROM (MS ROM)
4-Way Decode
Complex Decoder
Simple Decoder
Stack Engine (SE)
Decoded Stream Buffer (DSB) (µOP Cache) (1.5k µOPs; 8-Way) (64 B window)
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 µOPs)
Micro-Fusion
64B/cycle
Register Alias Table (RAT)
Branch Order Buffer (BOB) (48-entry)
Move Elimination
Rename / Allocate / Retirement ReOrder Buffer (224 entries)
Ones Idioms
Zeroing Idioms
Integer Physical Register File (180 Registers)
Scheduler Unified Reservation Station (RS) (97 entries)
Vector Physical Register File (168 Registers)
Port 0
Port 1
Port 5
Port 6
Port 2
Port 3
Port 4
Port 7
INT ALU
INT DIV
INT Vect ALU
INT Vect MUL
FP FMA
AES
Vect String
FP DIV
Branch
INT MUL
Bit Scan
Vect Shuffle
LEA
AGU
Load Data
Store Data
EUs
Execution Engine
L2 Cache 256KiB 4-Way
Unified STLB
32B/cycle
To L3
Store Buffer & Forwarding (56 entries)
Load Buffer (72 entries)
L1 Data Cache 32KiB 8-Way
Data TLB
Line Fill Buffers (LFB) (10 entries)
Memory Subsystem

Figure 1: Intel skylake microarchitecture diagram.

- *Speculative execution:* Instruction are executed speculatively. This means in particular that the CPU is able to guess which path of execution the program will take and execute preemptively instructions while at the same time checking if the path taken was the correct one. This can reduce execution time as latencies fetching memory addresses (needed to evaluate a branching condition for example) can be hidden by going forward without waiting on the memory subsystem sending bytes back. Modern CPU possesses branch predictors that are able to guess pattern based on previous behavior. However, programs with unpredictable logic do not benefit much from the branch predictor as misspredicting a branch requires the whole pipeline to be thrown out and execution to be resumed to a previous state. Thus, it is the programmer's responsibility to optimize a program to accomodate for the branch predictor: *e.g.* doing branchless programming whenever possible.

- *Out of order execution:* Modern processors can schedule instructions out of order, meaning that up to denpendencies, they are able to execute instructions however they see fit to better utilize the core's resources. Modern CPU have a *front end* responsible for decoding instructions (for instance x86-64 instructions) into μ-ops (proprietary CPU-dependant lower-level instruction set). Those μ-ops are then scheduled to saturate the execution units and reduce stalls and idle cycles.

- *Hyperthreading/simultaneous multithreading (SMT):* Hyperthreading allows two or more threads to have their stack and program counter on the same core but share computational resources. This allows the CPU core to quickly switch between threads to hide memory latencies.

- *A cache hierarchy:* As memory speeds have nearly reached their physical limits, a hierarchy of caches is used for better throughput and latency. They serve as buffer spaces to store regularly accessed memory closer to the execution units (and thus reducing latency). However, the closer the cache is to the core, the smaller it is. Thus, a greater emhpasis on optimizing for size and data access patterns can lead to better performance.

All of those optimization take away precious die space (that could have been used towards compute capabilities) but it is the price to pay have a CPU that can handle all types of workloads decently well. Having said that, in recent years, the emphasis has been put on increasing throughput as well by increasing the core count of a CPU and relying on:

- *Instruction level parallelism (ILP):* Modern CPU use *instruction piplelining* allowing multiple instructions to be overlapped by having multiple steps of the pipeline (*e.g.* instruction fetch, decode, execution, memory writes) running concurrently like an assembly line. Together with the scheduler, this can increase the throughput by being able to execute multiple independant instructions simultaneously.

- *data level parallelism:* Modern processors have *Single Instruction Multiple Data* (SIMD) instructions that operate on vector of elements instead of scalars increasing dramatically throughput. Contrary to previous techniques, vectorization (*i.e.* the act of leveraging those instructions) is not done automatically but requires either using an optimizing compiler able to generate those special SIMD instructions or using a programming model that encapsulates those instructions (*e.g.* using Intel intrinsics or OpenMP).

More than ever, the world of computing and HPC is focusing more and more on many-cores architectures (instead of increasing clock frequency which have reached a wall) with complex memory hierarchies (to have memory the closest to computation to optimize memory throughput) and the use of accelerators such as GPUs (presented in the next subsection) to gain in efficiency.

2.2 GPU architecture basics

Contrarily to CPUs, GPUs are not tailored towards general purpose computation with arbitrarily complex program logic but focus on compute intensive workloads. This is achieved thanks to a simpler core design. There is no branch predictor, no out of order execution, fewer caches. Thus a higher percentage on the die is dedicated to arithmetic computation. Moreover, GPUs make extensive use of SIMT (*Single Instruction Multiple Threads*): threads of execution execute concurrently in groups of 32 called warps.

We will detail the architecture of the Nvidia RTX A6000 GPU (2). It is composed of 84 *streaming multiprocessors* (SM). Each streaming multiprocessor has 4 schedulers each able to schedule one instruction per cycle to be executed on a warp (group of 32 execution threads). It also has multiple caches for instructions, a register file used to store the stack of each thread of execution running (*i.e.* the variables defined in the program running on the GPU). Most of the space remaining is dedicated towards single precision and double precision arithmetic. Shared between all streaming multiprocessors is a L2 data cache, a PCIe express to interface with CPU memory and multiple memory controllers to access GPU global memory. It also has a global scheduler that distributes the workload onto the streaming multiprocessors.

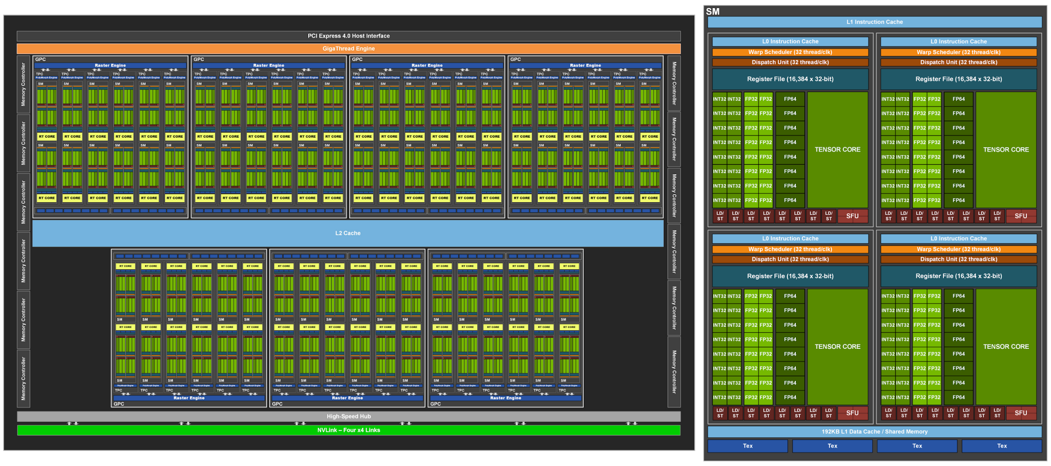


Figure 2: Nvidia RTX A6000 chip diagram (left), Streaming Multiprocessor diagram (right).

At this point, it is important to detail the GPU execution model to be able to better understand how the GPU distributes and execute a workload.

2.2.1 GPU execution model

There exists multiple GPU execution models. The most common ones are CUDA (stands for Compute Unified Device Architecture), and openCL (open Compute Language). Those two models are very similar as only the terminology changes. We will focus here on CUDA as this is the one that will be used. To fit the extreme parallelism of the GPU, a SPMD (Single Program, Multiple Data) paradigm is employed. This means that the same program is run on all threads of execution but deal with different data. Thus, there is no explicit

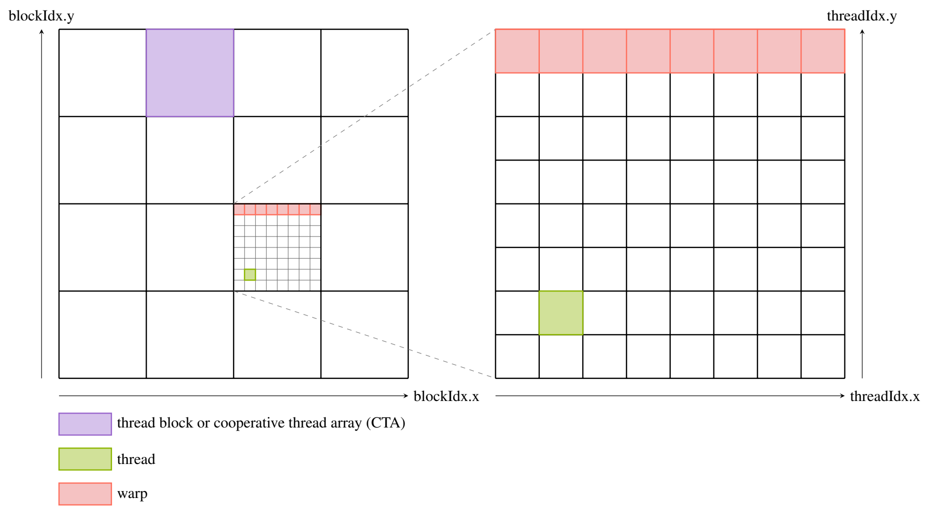


Figure 3: Example 2D kernel grid.

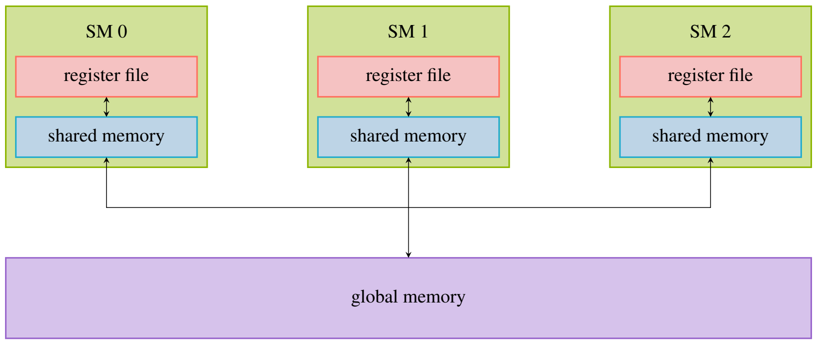


Figure 4: GPU memory hierarchy.

outer parallelized loop (this is the same approach used by MPI). Those program are called GPU kernels and are executed on a grid hierarchy. Each element of the grid is called a thread, and each thread can retrieve its position within the grid using implicit variables and then use its position to fetch specific data. Grids can either be 1D, 2D and 3D to better fit the use case. Grids are arranged in block of threads (also called cooperative thread arrays in CUDA or work groups in openCL terminology). Within the same block, threads are able to share data between them as well as synchronize. However, no synchronization between threads belonging to different blocks is possible. This allows the GPU to schedule block independently on different streaming multiprocessors. Thus, algorithm have to take into account this restricted level of communication that allows greater parallelism. Furthermore, within a block, groups of 32 threads called warps (or wave in openCL terminology) execute instruction in lockstep. When a kernel is queued up to be executed, the scheduler distributes blocks to multiple streaming multiprocessor (streaming multiprocessors are called compute units in openCL). Depending on the resources needed for a block, a streaming multiprocessor is able to run multiple blocks at a time. This oversubscription of blocks (and thus warps) per streaming multiprocessor/scheduler allows the GPU to hide latencies by letting the 4 schedulers switching to different warps within the same block or of a different block scheduled to the SM while waiting on memory, synchronization primitives, instruction fetch etc. This is made possible because there is not cost associated to switching contexts between different warps. Therefore, it is recommended to tune the block size to maximize the occupancy of the SM (*i.e.* the number of warps at a given time that the SM can schedule). For the RTX A6000, each scheduler can hold a maximum of 12 warps, so each SM can hold at most 48 warps.

2.2.2 GPU memory model

GPU have different memory spaces. Contrarily to CPU programming, the GPU memory model exposes to the programmer different memory spaces. CUDA distinguishes 4 memory spaces:

- *Global memory:* This constitutes memory accessible to all threads in a grid. This is the slowest but largest memory space on the GPU. Moreover, this is the only memory space that is retained between kernel invocations.
- *Shared memory:* This is memory located in the streaming multiprocessor that can be used to shared data between threads of the same thread block. This is a faster memory space that can be used to cache global memory that may be used multiple times by different threads of a thread block.
- *Register:* This is memory accessible per thread located in the register file. This constitutes where the stack local variables of a kernel are stored if there is enough space.
- *Local memory:* (not to be mistaken with shared memory) This is global memory that can be used when too many variables are in use and cannot be all stored in the register file.

2.3 GPU programming paradigms

Here is a short survey of the most common GPU programming models:

- The CUDA platform is able to seamlessly integrate GPU code as well as CPU (called host code). As such, source files in CUDA C++ can both contain regular C++ functions and classes targetting the CPU and GPU kernels as well as code that can be ran on both CPU and GPU. This ease of use together with the high adoption of CUDA makes it a great choice to program on GPUs. The only downside is that with CUDA, you are tied to the Nvidia ecosystem of GPU as you cannot compile for other GPUs.
- OpenCL (Open Compute Language) is a framework initiated by the Khronos group for offloading computation using a similar programming model than CUDA. It uses C/C++ on the host side and either OpenCL C or OpenCL C++ for device code (device here means GPU). The advantages of OpenCL is that it is an open standard that can target many devices: CPUs, GPUs, FPGA. However, the need to separate device code from host code is error prone (as no type checking can be done between device and host code) and the compilation toolchain is rather complicated.
- SYCL is an effort by the Khronos group to make a single source programming abstraction using openCL as a backend.
- KOKKOS is a C++ library initiated by the US department of Energies Exascale Project that can target multiple backends: CUDA, SYCL, OpenMP, C++ threads. It heavily relies on C++ templates and is designed to target supercomputers. Both SYCL and KOKKOS exposes a higher level abstraction than CUDA and OpenCL making it easier to port an existing codebase using those libraries.
- OpenMP and OpenACC are two programming models that rely on compiler directives to offload computation.

In this report, we have chosen to use CUDA for its ease of use and lower-level capabilities.

2.4 GPU best practices

In this section, we will describe a few optimization concepts specific to GPUs.

2.4.1 Kernel level

- *warp divergence:* As stated previously, groups of 32 threads are executed in lockstep in a SIMT fashion. Therefore, if threads in a warp go through different code paths, the GPU needs to go through all codepaths that the threads in a warp visit. The way this is done is using bitmasks: when threads in a warp encouter an if statement, the GPU construct a bitmask of the `if` condition. If all threads in a warp visit the same branch, only the instructions of this branch are executed. However, when the bitmask contains both zeros and ones, the GPU masks off the lanes taking the `else` branch and executes the instruction in the `if` branch (the masked lanes are idle) and then masks of the lanes taking the `if` branch and executes the instructions in the `else` branch. This means that the instructions in the `if` and `else` branch are being serialized. This results in poor GPU utilization (as the lanes masked off do not contribute to advancing further the algorithm). Furthermore, nested `if` conditions reduces GPU utilization even more. It is then recommended to try whenever possible to have all threads in a warps take the same path to avoid needing to execute all paths. Figure 5 illustrates a warp divergence.

```
if ((threadIdx.x % 32) < 16) {
  A;
  B;
} else {
  C;
  D;
}
E;
```

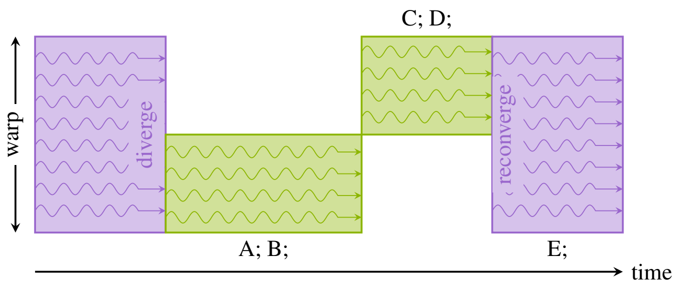


Figure 5: warp divergence: on the left, an example kernel where each warp branches depending on the lane index and on the right a representation of the execution of such a kernel.

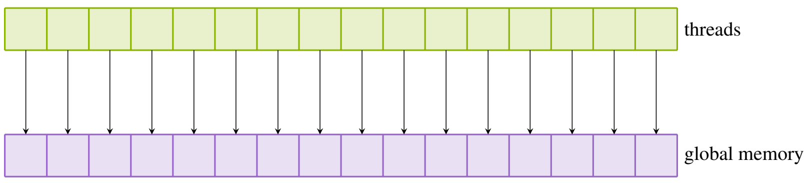


Figure 6: coalesced global memory access.

- *Coalesced global memory accesses:* Accessing global memory in a kernel is relatively slow. Therefore, it is recommended to minimize the number of transactions. To do so, prefer accessing consecutive memory addresses from adjacent threads (this is called coalescing) and avoid strided accesses. It is then recommended to favor structure of arrays (SoA) instead of arrays of structures (AoS) to avoid strided accesses to global memory.

- *Avoid bank conflicts:* A great way to reduce accessing global memory is to cache data to shared memory for faster retrieval. The shared memory is composed 16 memory banks. If each half warp fetch shared memory in different banks, only one transaction is required. Otherwise, there is a bank conflit (see figure 7): and shared memory accesses are serialized within memory banks. There is however one exception: if every thread in a warp fetch the same address, a broadcast happens and only one transaction is needed (rather than having a 16-way bank conflict). For that reason, it is recommended to favor reads to shared memory that minimize bank conflicts.

- *Optimize kernel launch parameters:* As stated before, a GPU kernel is launched on a grid hierarchy. To better utilize the GPU, we need to oversubscribe the streaming multiprocessors as much as possible. To do so, the grid and block size must be carefully chosen. As blocks are distributed among streaming multiprocessors, the block size must be ajusted so that as many block as possible can fit in the resources of a streaming multiprocessor. The maximum occupancy of a kernel (*i.e.* the maximum of warps that a SM can handle at a given time) depends on multiple factors: number of registers used, shared memory, block size. As such, optimization tools provide us with graphs representing the maximum theoretical occupancy of a given kernel with respect to those factors (see figure 8). Moreover, the grid size must also be taken into account. A grid size too small might not keep the GPU fed. Blocks are distributed to SMs in waves. Ideally, a grid should be divided in a certain amount of full wave (each wave containing as many block as needed to fully occupy every streaming multiprocessor) to utilize the whole GPU. A partial wave at the end might be needed if the grid cannot be divided exactly into full waves but it should be avoided as it underutilize GPU resources.

- *Maximize floating point performance:* A good indicator of a kernel's performance is the number of floating point operation that it does per seconds. To judge a kernel's performance, the roofline model is often used. On a roofline model, kernels are plotted in the space of arithmetic intensity (*i.e.* the number of operations per byte) in abscissa and FLOPS (*i.e.* the number of floating-point operations per second) in ordinate. The maximum FLOP rate is plotted as a roofline and depends on the device used. For kernel with low arithmetic intensity, we are limited by the bandwidth of the GPU: too few operation are done compared to the memory needed, the GPU is starving for data, we are *memory bound*. On the contrary, when the arithmetic intensity is high enough, we can reach the peak FLOP rate of the device: the kernel is *compute bound*. Ideally, to fully utilize the memory bandwith and compute

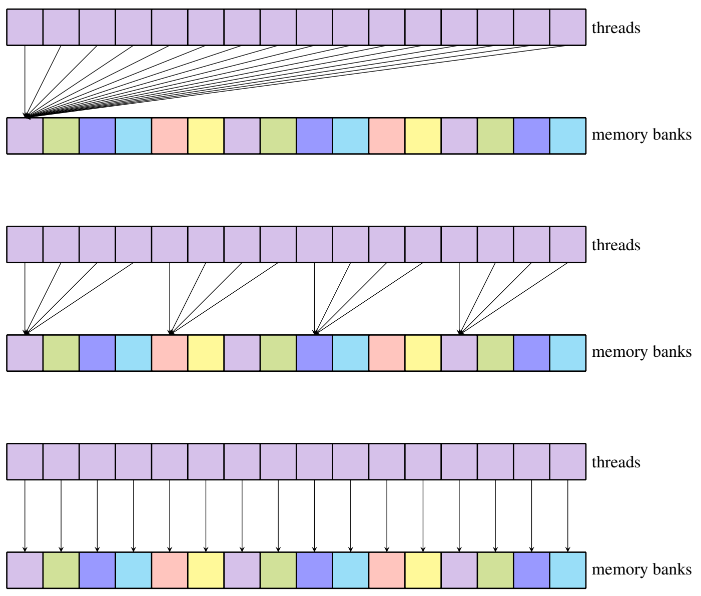
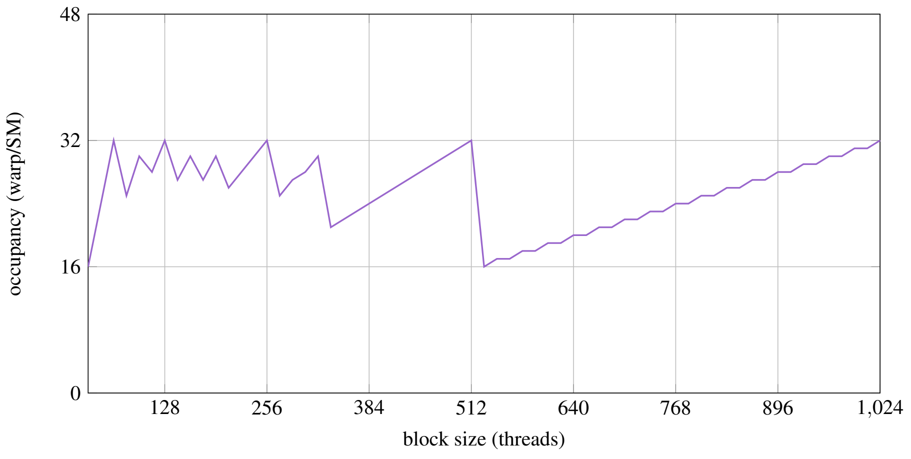


Figure 7: **top**:no bank conflict, **middle** 4-way bank conflict, **bottom**: broadcast.



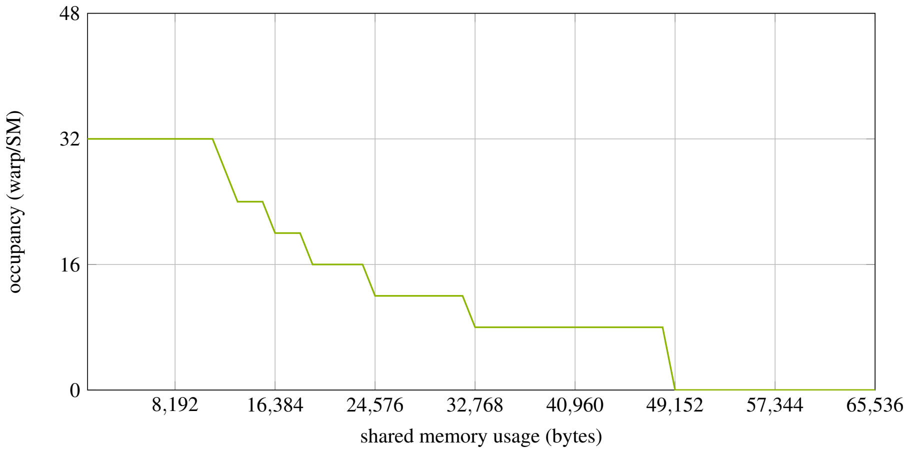


Figure 8: maximum theoretical occupancy of a kernel with respect to block size and shared memory utilization.

capabilities of the device, a kernel should aim at the limit between being compute bound and memory bound where both the memory bandwidth and compute power is fully utilized. In practice, this is hard to achieve. Moreover, a kernel should be as close as possible to the roofline. However in practice, for non trivial workloads, other factors than the arithmetic intensity can hinder performance: such as

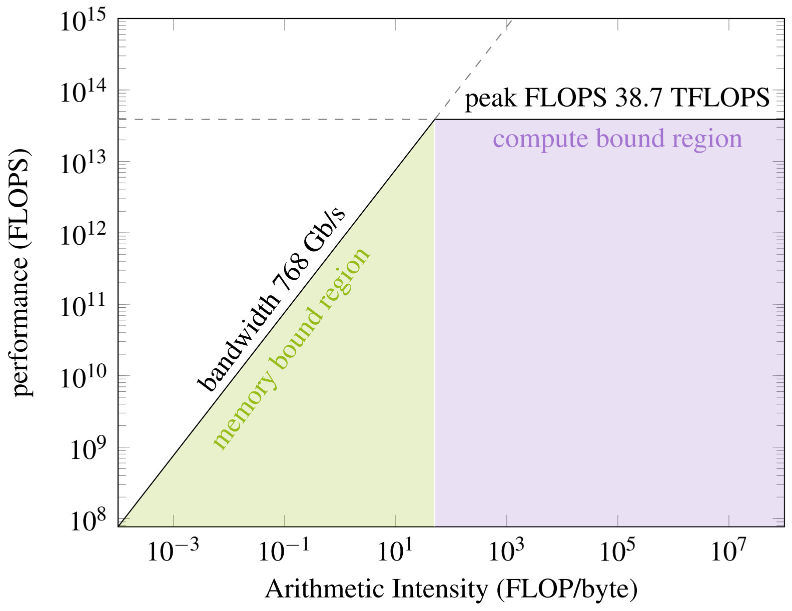


Figure 9: roofline model for the RTX A6000 in floating point single precision.

synchronization, branching. Figure 9 is a roofline model of the Nvidia RTX A6000 that we will use showing its performance characteristics in single precision.

2.4.2 Application level

Once the GPU kernels have been optimized, we can focus on whole program optimization. One of the most important aspect to think about is when to do memory transfers. It is recommanded to try to overlap host to device and device to host memory transfers with computation as the memory bandwidth between CPU and GPU is low. This can be done using CUDA asynchronous API calls allowing the CPU to do computation while the transfer is being done (instead of blocking the CPU waiting for the transfer to be done). Moreover, independant kernels and memory transfers should be run concurrently as modern GPUs are able to do computation tasks and memory transfers simultaneously.

Moreover, storing intermediary results to memory for later use in another kernel can lead to worse performance than doing the same computation twice. Indeed, as reading from global memory is slow, computing the same value twice instead of fetching memory leads to greater arithmetic intensity as less memory bandwidth is necessary and more computation is done. As reaching the peak FLOPS is in practice difficult, avoiding storing intermediary results counterintuitively in most cases leads to better performance. For the same reason, it is preferable whenever possible to group multiple kernels working on the same CUDA grid into one big kernel to avoid storing intermediary results to global memory (that was used to be retrieved intermediary results in later kernels). Doing so reduces global GPU memory usage and bandwidth. This has the added benefit of reducing the GPU scheduling overhead. This can lead to much better GPU utilization at the cost of violating programming best practices: one kernel has now multiple possibly unrelated concerns that were expressed before in multiple indivisible kernels.

I will consider all of those optimization aspects when building my library.

3 Adaptive mesh refinement

In this section, we will briefly introduce different techniques of adaptive mesh refinement and then focus on detailing the t8code library.

3.1 A quick survey of dynamic mesh adaptation techniques

Adaptive mesh refinement is a technique used in numerical simulations to adapt the accuracy of a solution in certain regions of interest. This is especially useful in fields tackling problems where features at different scales can appear: in astrophysics, hydrodynamics, acoustics to track wave fronts for instance. During the simulation, mesh elements can be refined in place where more accuracy is needed or derefined. The advantages of adaptive mesh refinement are computational and storage savings compared to static grids, a finer control of the grid resolution using a fine-tuned refinement criteria. However, using adaptive mesh refinement adds a layer of complexity and performance cost associated to the refinement procedures. When dealing with simulation running on multiple cores or distributed memory architectures, load balancing plays a significant role in an efficient implementation. Moreover, mesh refinement may introduce more singular grids containing hanging nodes or highly deformed mesh elements that require developping a solid numerical scheme that can handle those types of meshes.

There exists multiple approaches to adaptive mesh refinement:

- *Unstructured AMR:* This type of adaptation deals with unstructured meshes and is able to refine a mesh keeping the mesh conformal.
- *Block-structured AMR:* This type of adaptation works on a hierarchy of overlapping patches of structured grids. With this type of adaptation, there is less fine control over the refinement compared to other types of AMR but it has many advantages: it is easier to port an existing code for structured grids to use AMR. Substepping can also be used to reduce computation: the timestep depend on the patch mesh element size, bigger element uses bigger timesteps. It can also have better performance as there is less book-keeping and data access patterns are more predictable. For those reasons, this is the type of AMR that is best suited for GPUs. AMReX [19] is an example of such a block-structured AMR framework targetting modern HPC system architectures.
- *Tree-based AMR:* This type of AMR uses a quadtree or octree data structure to partition the spatial domain. Refining a mesh element consists in adding children nodes to the node representing the element and coarsening consists in discarding children of a node. This data structure can also be layed out linearly in memory using space filling curves ensuring good cache utilization. To deal with more complex domains, a forest of trees can be considered. P4est [3] and t8code [10] are two highly scalable tree-based AMR libraries. In this work, we will use the latter.

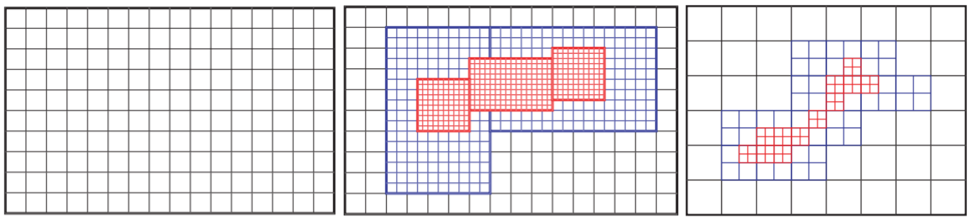


Figure 10: regular grid (left), block-structured AMR (center), tree-based AMR (right).

GPUs can be used in two manners for AMR: either the mesh adaptation is done on the CPU and the GPU only computes numerical scheme stencils, or both the AMR and stencil computation are done one the GPU. The former approach is the most common as it is the easiest to implement and mesh adaptation routines are not well suited for GPU because of their complex logic. Moreover, most AMR libraries targetting GPUs use a block-structured approach [16, 18, 1]. Nevertheless, some codes developed to do adaptation on the GPU using tree-based AMR for specific problems do exist [5, 13] but no generic frameworks have been developed to my knowledge.

3.2 T8code

T8code is a software library for scalable adaptive mesh refinement that has been shown to scale to over one trillion mesh element [9] using up to one million parallel processes [8]. T8code starts with a base unstructured mesh called coarse mesh. This mesh can contains different element types: tetrahedron, pyramids, hexes. On each coarse mesh element, it constructs a tree structure to be able to refine each coarse mesh element further. The collection of forests are then distributed and form a forest of trees (see figure 11). One of the strength of t8code is that it is able to do tree-based AMR on element types different than quads in 2D and hexes in 3D. To do so, it separates high-level (mesh global) algorithms from low-level (element) implementation. The low-level constructs thus depend on the element type (pyramid, hex, tetrahedron) and the high-level algorithms do

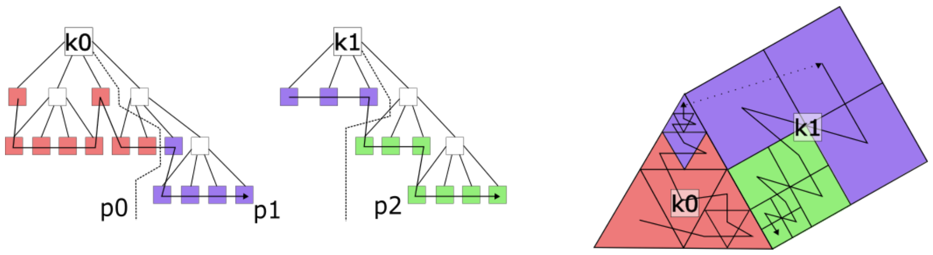


Figure 11: Example of a t8code forest on the left containing two trees distributed over 3 processes and on the right the associated mesh qnd order of traversal of the mesh element along the space filling curve. By courtesy of Johannes Holke taken from [10].

not depend on the different element types. This makes the library extendable to many element types easily. Moreover, space-filling curves had to be developed to handle pyramid and tetrahedral [2] element types. Data associated to element is stored linearily using the ordering induced by the space filling curves per tree (see figure 11). Moreover, these curves possess fundamental properties necessary for the mesh adaptation routines and have the added benefit that neighboring elements have data associated to them close to each other increasing performance with good cache locality.

The t8code library relies on user provided callback functions to handle mesh refinement and coarsening. To do so, the API exposes a few important routines:

- *New:* This operation constructs a uniformly refined mesh from a coarse mesh supplied by the user distributed across parallel processes.
- *Adapt:* Given a callback function, this procedure decides which element need to be refined and coarsened.
- *Balance:* This procedure ensure that neighboring elements have a refinement level different no bigger than 1. This 2:1 balance condition might be necessary for certain numerical schemes.
- *Interpolate:* This procedure allows the user to interpolate variable data from one tree to its refinement.
- *Partition:* This procedure together with the *partition data* procedure redistributes the elements accross parallel processes to balance the load. Custom weights per element can also be provided to have a fine control over the mesh partition.
- *Ghost:* and ghost related operations allow the user to fetch data from ghost elements that are owned by other processes.
- *Iterate:* This procedure allows the user to iterate through the mesh and request face neighbor information as well.

Figure (12) shows how an application can use t8code for a numerical simulation loop. In order to customize the refinement behavior and interpolation, the user needs to provide callback functions.

4 T8gpu

T8gpu [11] is the name of the numerical simulation framework that I developed. It is a header-only C++17 library using CUDA relying heavily on C++ template metaprogramming to provide an easy to use API but compiles down to optimized code. A first framework to build finite volume scheme was developed that can handle hybrid meshes with different element types. As this naive approach is not efficient, a second subgrid approach was developped working on quad and hex elements which proved to be more efficient as the simulation loop is better suited towards GPU computations.

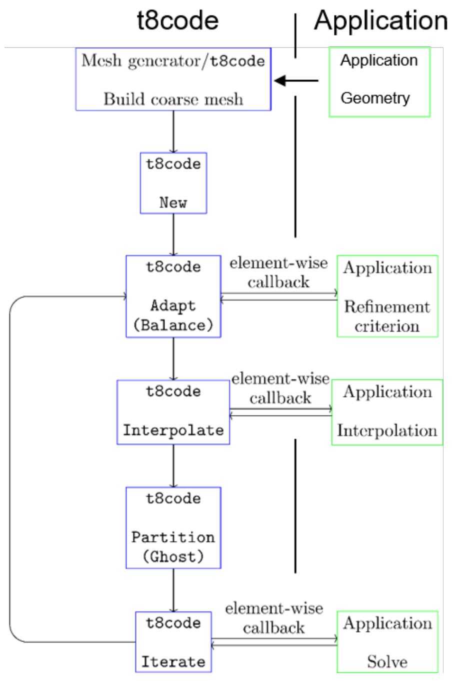


Figure 12: flowchart of a typical utilization of t8code. By courtesy of Johannes Holke taken from [10].

4.1 Challenges associated to AMR and the CPU/GPU model

One of the biggest hurdles on hybrid CPU/GPU AMR is communication between CPU and GPU. On most system, data transfers are done through the PCIe bus. This bus can have a bandwidth up to 64Gb/s. However, this pales in comparison with the 768Gb/s bandwidth that the RTX A6000 has between its global memory and streaming multiprocessors. Thus, minimizing data transfer and/or hiding transfer by overlapping computation is necessary to garantee adequate performance.

Moreover, as tree-based AMR offer fine grained control over the refinement and coarsening of mesh elements, mesh adaptation routines take more time per mesh element compared to block-structured approaches. That is why the subsequent idea of using subgrid element to mimic certain block structured AMR methods proves to be efficient: the amount of computation on the GPU per mesh element is greatly increased making the mesh adaptation and data transfers manageable. This subgrid approach also has better GPU data access patterns.

Another challenge is that the t8code library distributes the mesh among processes. Therefore, it is necessary to communicate between multiple CPU processes but also to one or more GPUs from multiple processes.

4.2 Design considerations

In order to use a GPU on a multiprocessing program, two choices can be made regarding managing the GPU:

- One option is to have each process launch kernels to do computation on the mesh elements that the process owns. The advantage of this approach is that it is symmetric and aligns nicely with t8code's mesh distribution. However synchronization between kernels launched by different processes needs to be taken care of.

- The other option is to have a main process that aggregates every GPU computation from all other processes into one kernel launch from this main thread. The advantage of this approach is that the synchronization is easier. Having only one big kernel launch instead of one kernel launch per process can results in better GPU utilization as well. However this approach may lead to CPU load imbalance between the main process and the other processes.

For our library, I have chosen to implement the first option because it copies more closely t8code's data management to the GPU.

4.2.1 GPU memory handling

To easily handle GPU memory, I have devised a C++ class that encapsulates GPU memory allocation that can be shared between processes. Contrary to the distributed memory model between CPU processes, as we for now only use one GPU, we have decided to share GPU memory between CPU processes resulting in better GPU memory usage (as we do not need to have ghost element and copy memory around to handle interfaces between processes). To do so, we created the `t8gpu::SharedDeviceVector<T>` vector class templated on the element type `T`. This class lets the user allocate GPU memory from every process with their own size (ideally to fit information related to the process owned mesh elements). Moreover, each process can retrieve the memory allocated by the other processes to access data associated to ghost mesh elements. To access GPU memory allocated by another CPU process, an address translation needs to be done using IPC (inter process communication). As its name implies, the `t8gpu::SharedDeviceVector` can be resized to accomodate for mesh repartitioning, adaptation. However, contrarily to the `std::vector`, resizing discards the previous data allocated. Thus the user needs to explicitely copy information from the previous sized allocation to the new allocation. This is intentional to force the user to correctly copy data (by projecting variable data when adapting a mesh for example). This class is the backbone of t8gpu and is used everywhere to more easily handle GPU memory allocation.

An overload of this class for the `std::array<T, N>` array element types has been implemented to convert the data layout from an array of structures (in this case the structure being an `std::array<T, N>`) to structure of arrays for better data access patterns. This overload provides an interface to access specific elements within the vector of arrays. This array of structures to structure of array transformation (see figure 13) is crucial because it allows GPU kernel memory access into such a vector to be coalesced when accessing neighboring element data. Figure (14) represents how the class access GPU memory allocation from other processes for 3 MPI processes. Similarly to `std::vector`, the resize method allocates extra space (usually 1.5 times the requested size) to avoid allocating GPU memory often. One of the downside of this class is that resizing requires global communication of all processes to translate the new allocations into process local dereferencable pointers. That is why, to minimize communication, all further classes use only one shared vector of a struct element type containing all the data per mesh element (we use exclusively the optimized overload for the `std::array<T, N>` type).

Furthermore the `t8gpu::MemoryManager` template class encapsulates the `t8gpu::SharedDeviceVector` class providing a better interface for numerical simulation codes. This class is templated on an `VariableType` enum class containing the name of the variables of interest at the mesh elements and another enum type called `StepType` enumerating the names of the timestepping substeps (this allows the user to get as many storage for variables as there is substeps). This allows the user to implement multiple timestepping routines: explicit Euler, Runge-Kutta with ease and reference variable data by their name making this class more convenient. By specializing the `t8gpu::variable_traits` struct on a `VariableType` enum class, the user is able for now to switch from using double precision or single precision. In the future, this `t8gpu::variable_traits` approach can be used to customize further the behavior of t8gpu and make it more extensible.

For instance, to deal with the compressible Euler equations in 3D using 3rd order Runge-Kutta timestepping, you can define:

```
#include <t8gpu/memory/memory_manager.h>

enum VariableList {
  Rho,      // density.
  Rho_v1,   // x-component of momentum.
  Rho_v2,   // y-component of momentum.
  Rho_v3,   // z-component of momentum.
  Rho_e,    // energy.
  nb_variables // number of variables.
};

enum StepList {
  Step0,    // used for RK3 timestepping.
  Step1,    // used for RK3 timestepping.
  Step2,    // used for RK3 timestepping.
  Step3,    // used for RK3 timestepping.
  Fluxes,   // used to store face fluxes.
  nb_steps // number of steps.
};
```

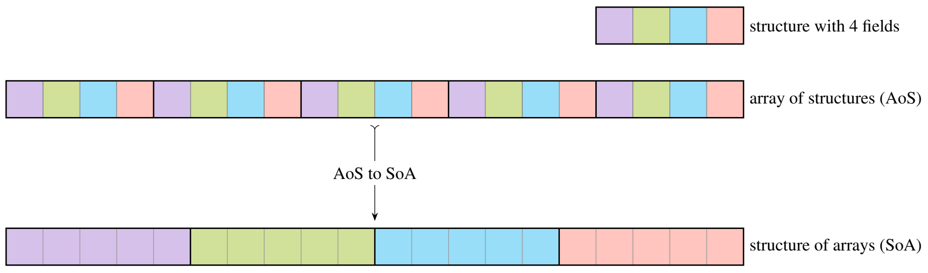


Figure 13: Array of structures to structure of arrays transformation.

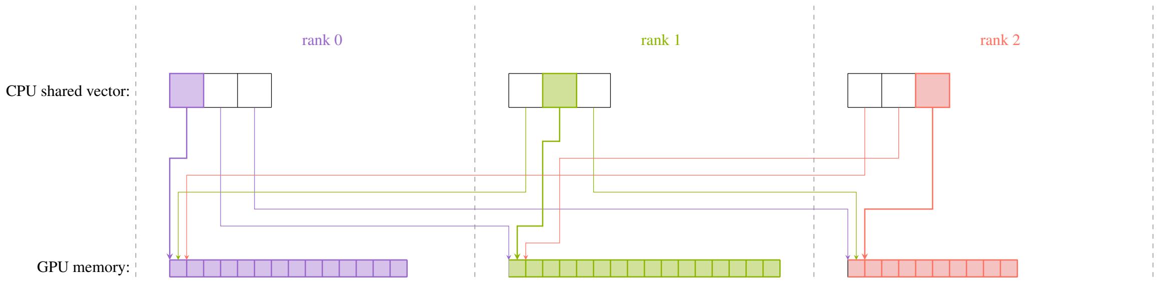


Figure 14: shared vector data layout diagram for 3 MPI processes.

```
// We modify the MemoryManager class behavior by specializing this trait struct:
template<> struct t8gpu::variable_traits<VariableList> {
  using float_type = double;      // we explicitely select double precision.
  using index_type = VariableList // we use VariableList enum to select variable.
  static constexpr size_t nb_variables = VariableList::nb_variables;
};

// This class is then tailored to our use case:
using MemoryManager = t8gpu::MemoryManager<VariableList, StepList>;
```

4.2.2 Mesh management

The mesh management is done through the template class `t8gpu::MeshManager` class that takes ownership of a t8code mesh. This class encapsulates a `t8gpu::MemoryManager` class as well to store variable data per mesh element. It provides an API to do mesh adaptation by allowing the user to implement a mesh refinement criteria and provides many CPU and GPU helper function to query mesh connectivity information. This class provides a good level of flexibility allowing the user to be able to compute mesh refinement criterion on the GPU, compute face fluxes and do timestepping.

As t8gpu distributes the mesh among processes (see figure 15) but t8gpu uses a GPU shared memory approach, some extra bookkeeping is needed to reconciliate both approaches. To do so, the `t8gpu::MemoryManager` class stores MPI rank information and index information for both process owned mesh element and process ghost mesh elements. This allows the user to seamlessly fetch mesh data for owned elements and remote (ghost) element in an uniform fashion (see figure 16). This is especially needed when iterating over faces: as they are distributed among process, some face at the interface between the domain of two processes need access to data associated to other processes.

Likewise, similar rank and index correspondance arrays are used for the projection procedure: instead of projecting from the previous mesh to the new mesh on the CPU, we send correspondance arrays (*i.e.* from the perspective of the new mesh, the correspondance array gives the index and ranks of the elements from the previous mesh it needs to project from) so that the projection can be done on the GPU. This minimizes the amount of data sent from and to the GPU (especially when dealing with subgrid elements in the next subsection).

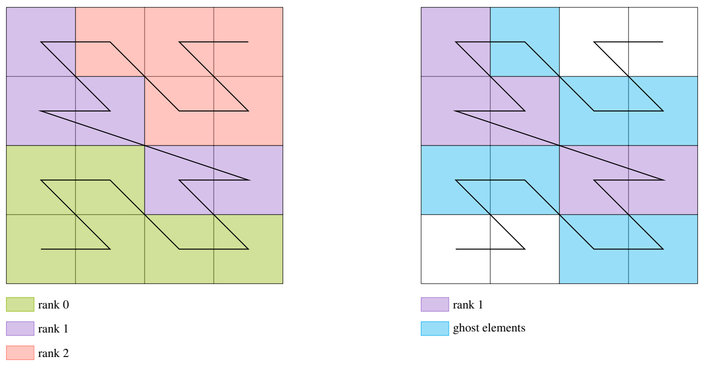


Figure 15: Mesh distribution along z-curve on one tree with uniform refinement to level 2 on 3 processes (left), mesh owned by the second process and its ghost layer elements.

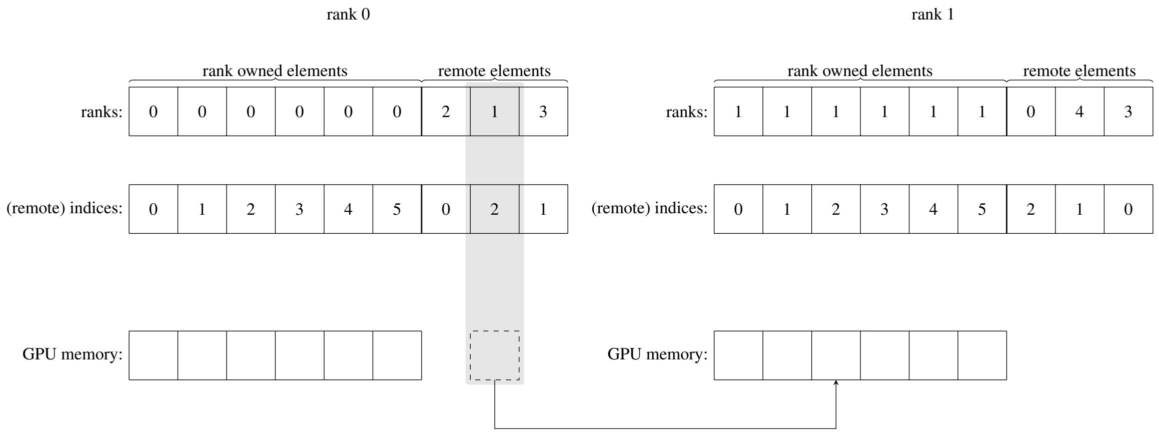


Figure 16: Ghost layer bookkeeping example.

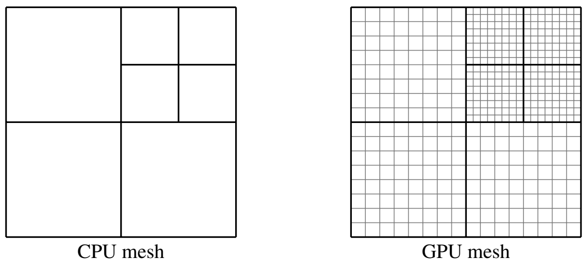


Figure 17: CPU and GPU mesh with 8×8 subgrid mesh elements.

4.2.3 Subgrid elements

As seen in the previous subsection, a lot of overhead is associated with mesh connectivity information. Moreover, as the GPU architecture is highly parallel, we expect an numerical simulation application using t8gpu and AMR to spend most of its time in the CPU mesh refinement routines. To remedy that, we have deciced to use subgrid elements to increase the ratio of GPU computation relative to CPU mesh transformation operations. By subgrid elements, we mean each mesh element on the CPU is considered to be a coarse mesh on the GPU and is further refined uniformly on the GPU (see figure 17). This requires additional bookkeeping (notably extra face neighbor mesh elements orientation information), but we expect the greater among of computation done per CPU mesh element to conteract the extra overhead. Another advantage of subgrid elements is that data access patterns can be made better. Indeed, in a kernel where each consecutive thread access the same variable at neighboring subgrid mesh elements, using a structure of arrays (where each array stores one variable for all mesh elements) to store variable data in column major order results in coalesced memory accesses. Moreover, choosing GPU kernel block to be the size of a subgrid is highly advandageous: it simplifies development and allows the user to use shared memory effectively to cache variable data.

To work with subgrid elements, the template classes `t8gpu::SubgridMemoryManager` and `t8gpu::SubgridMeshManager` have been implemented. As per the none subgrid equivalent classes, they are templated on a `VariableType` and `StepType` user-defined enum classes and an extra `SubgridType`. This type must be an instance of the `t8gpu::Subgrid<int... extents>` parametric helper struct. This type describes the subgrid size and dimension and provides helper functions to access multidimensional data in column major order and retrieve subgrid information. Here is what a simple kernel iterating over all mesh elements looks like using subgrid elements:

```
using namespace t8gpu;

__global__ void set_momentum(SubgridMemoryAccessorOwn<VariableList, Subgrid<8, 8, 8>> variables) {
  int const e_idx = blockIdx.x;

  int const i = threadIdx.x;
  int const j = threadIdx.y;
  int const k = threadIdx.z;

  auto [rho_u, rho_v, rho_w] = variables.get(Rho_u, Rho_v, Rho_w);

  // element index
  //    ┌─┴─┐
  rho_u(e_idx, i, j, k) = ...
  //           └──┬──┘
  //     subgrid coordinates
  ...
}
```

This kernel showcases the ease of use of the library and results in optimal memory access patterns. T8gpu provides a high-level API while still being a no-cost abstraction layer. For now, subgrid are only implemented for hexahedral and quad meshes.

Another strength of t8gpu is its ability to allow the user to implement reusable generic schemes. For instance, the user can implement timestepping schemes routines independantly of the variable type simply by iterating over the variables. This reduces code duplication and results in less error-prone code. Only when necessary, the user can do computation on a specific variable refered by its name, for instance when computing fluxes.

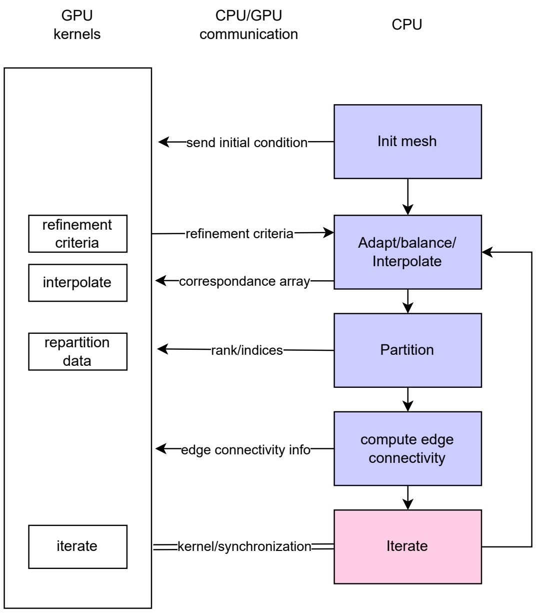


Figure 18: t8gpu flow diagram.

4.2.4 User workflow

From the perspective of an user, the typical application follows the workflow diagram (18). The mesh data structure is initialized on the CPU and the GPU can then compute the initial condition of the simulation. The main simulation loop begins by calculating a refinement criteria per mesh element on the GPU and send it to the CPU. Using this criteria, the mesh adaptation routines and balance can be run on the CPU and a correspondance array can be sent to the GPU containing the space filling curve indices of the new mesh elements. This information is enough to be able to project variable data from the previous mesh to the new mesh directly on the GPU. The partition CPU routine can optionally be used to balance the load on the CPU ranks either uniformly or by providing user defined weights. The new ranks and indices of of each cell of the newly partitioned mesh needs to be send to the GPU to be able to copy the necessary variable data from the previous partition to the new one. The next step consist in computing the face connectivity data and send it to the GPU to be able to correctly compute fluxes between mesh elements. Then multiple user defined iteration can be run on the GPU, needing only synchronization between ranks in-between iterations but no data transfers are needed and the next iteration of the loop can start. An example of this workflow diagram will be illustrated in the following sections.

4.3 Test cases

We are interested in the compressible Euler equations, which is the following system of hyperbolic conservation laws in 2D:

$$\begin{cases} \rho_t + (\rho u)_x + (\rho v)_y &= 0, \\ (\rho u)_t + (\rho u^2 + p)_x + (\rho u v)_y &= 0, \\ (\rho v)_t + (\rho u v)_x + (\rho v^2 + p)_y &= 0, \\ E_t + (u(E+p))_x + (v(E+p))_y &= 0. \end{cases}$$

The unkown quantities are the fluid density ρ, energy density E and the fluid momentum $\mathbf{m} = \rho\mathbf{v} = [\rho u, \rho v]^T$ as well as the pressure p. To close the system, we need one more equation. We use here the ideal gas law and to get $p = (\gamma - 1)\left[E - \frac{1}{2}(u^2 + v^2)\right]$ where $\gamma = 1.4$ is the specific heat ratio for a monoatomic gas.

4.3.1 Finite volume formulation

We have decided to implement a first order Godunov scheme using two different fluxes. The Godunov scheme can be derived from the following observation: for a 1D linear scalar conservation law, starting with a piecewise constant initial condition, for a small enough timestep Δt, the exact solution can be computed by piecing together analytical solution to Riemann problems at cell interfaces [12]. The idea of the Godunov scheme is thus starting from piecewise constant initial data, to evolve the hyperbolic equation exactly (or approximately) for one timestep and average the resulting function over the grid cells to get the new initial data used to compute the next timestep. This method can be generalised to multidimensional system of non-linear conservation laws as well. This provides a first order scheme in space and time but can be augmented to yield higher-order methods.

One way to get a higher order accuracy method in space is to use a linear or polynomial reconstruction instead of a constant reconstruction. One of the downsides of this approach is that such schemes are usually unstable and an artificial dissipation term needs to be added to get stability near shocks resulting in a lower order in those regions (the typical approach consists in blending a high-order scheme in smooth regions and the first oder Godunov scheme near discontinuities by introducing a dissipation term or flux-limiting).

The usual approach to get a higher order accuracy method in time is to use a method of lines approach: discretize in space first resulting in a system of differential equations and use a high order timestepping scheme such as the Runge-Kutta methods. This is the approach that we will take.

Spatial discretization: We can rewrite the compressible Euler equations in conservative form as:

$$\partial_t \mathbf{q} + \operatorname{div} \mathbf{f}(\mathbf{q}) = 0, \qquad \text{with:}$$
$$\mathbf{q} = \begin{bmatrix} \rho \\ \rho\mathbf{u} \\ E \end{bmatrix}, \quad \mathbf{f} = \begin{bmatrix} \rho\mathbf{u} \\ \rho\mathbf{u} \otimes \mathbf{u} + p\,\mathrm{Id} \\ \mathbf{u}(E+p) \end{bmatrix}.$$

Then, we integrate over each cell $K \in \mathcal{K}$ of the mesh and use the divergence theorem:

$$\frac{d}{dt}\left(\int_K \mathbf{q}\right) + \int_{\partial K} \mathbf{f}(\mathbf{q}) \cdot \mathbf{n} d\Gamma = 0.$$

We denote by $\mathbf{q}_K$ the mean value of $\mathbf{q}$ in K. We make the following approximation:

$$\frac{d\mathbf{q}_K}{dt} = -\frac{1}{|K|}\int_{\partial K} \mathbf{f}(\mathbf{q}) \cdot \mathbf{n} d\Gamma \approx -\frac{1}{|K|}\sum_{e \in \partial K} \int_e \hat{\mathbf{f}}(\mathbf{q}_K, \mathbf{q}_L) \cdot \mathbf{n} d\Gamma := L(\mathbf{q})_K. \tag{1}$$

where $\hat{\mathbf{f}}$ is the numerical flux: *i.e.* an approximation of the flux at the origin of the Riemann problem with left and right states $(\mathbf{q}_K, \mathbf{q}_L)$. We then get a system of non-linear differential equations.

Time discretization: We have chosen to implement an optimal third order strong stability perserving Runge-Kutta method [6] (or SSP-RK3). This scheme belong to a family of schemes for which the TVD (total variation diminishing) property holds:

$$TV(\mathbf{q}^{n+1}) \leq TV(\mathbf{q}^n), \quad TV(\mathbf{q}) := \sum_j |\mathbf{q}_{j+1} - \mathbf{q}_j|, \quad \text{in 1D.}$$

More generally, this property holds true for every semi-norm, not just the total variation. Therefore, these scheme preserve the stability: *i.e.* if the Euler forward scheme is stable, then the SSP-RK scheme are also stable. More specifically, these scheme are chosen so that they preserve this TVD property while maximizing the CFL condition for a linear numerical scheme with the same number of stages.

In general, a m-stage Runge-Kutta method for the differential equation $\frac{d\mathbf{q}}{dt} = L(\mathbf{q})$ can be written in the form:

$$\begin{cases} \mathbf{q}^{(0)} &= \mathbf{q}^n, \\ \mathbf{q}^{(i)} &= \sum_{k=0}^{i-1} \left(\alpha_{i,k} \mathbf{q}^{(k)} + \Delta t \beta_{i,k} L\left(\mathbf{q}^{(k)}\right) \right), \forall i \in [\![1, m]\!], \\ \mathbf{q}^{n+1} &= \mathbf{q}^{(m)}, \end{cases}$$

with $(\beta_{i,k})_{\substack{1\le i\le m \\ 0\le k<i}} \in \mathbb{R}^+$ and $(\beta_{i,k})_{\substack{1\le i\le m \\ 0\le k<i}} \in \mathbb{R}^+$ such that $\sum_{k=0}^{i-1} \alpha_{i,k} = 1$ for all $i \in [\![1, m]\!]$.
More precisely, if the Euler method is strongly stable under the CFL condition $\Delta t \le \text{CFL}$:

$$(|\mathbf{q}^n + \Delta t L(\mathbf{q}^n)| \le |\mathbf{q}^n|,$$

with $|\cdot|$ a semi-norm. Then, the Runge-Kutta method is SSP:

$$|\mathbf{q}^{n+1}| \le |\mathbf{q}^n|,$$

provided the following CFL condition is fulfilled:

$$\Delta t \le c \cdot \text{CFL}, \quad c = \min_{i,k} \frac{\alpha_{i,k}}{\beta_{i,k}}. \tag{2}$$

It can be shown that $c \le 1$ and the value $c = 1$ is attainable.
An optimal third-order SSP Runge-Kutta method that we will use is given by:

$$\begin{cases} \mathbf{q}^{(1)} &= \mathbf{q}^n + \Delta t L(\mathbf{q}^n), \\ \mathbf{q}^{(2)} &= \frac{3}{4}\mathbf{q}^n + \frac{1}{4}\mathbf{q}^{(1)} + \Delta t L(\mathbf{q}^{(1)}), \\ \mathbf{q}^{n+1} &= \frac{1}{3}\mathbf{q}^n + \frac{2}{3}\mathbf{q}^{(2)} + \Delta t L(\mathbf{q}^{(2)}), \end{cases}$$

where $\mathbf{q} = (\mathbf{q}_K)_{K\in\mathcal{K}}$, $n \in \mathbb{N}$ is the current iteration and $L(\mathbf{q}) = (L(\mathbf{q})_K)_{K\in\mathcal{K}}$ is defined in (1). The CFL coefficient for this scheme is $c = 1$ defined in (2).

4.3.2 The HLL and KEPES fluxes

The HLL (for Harten, Lax and van Leer) approximate Riemann solver introduced in [7] is derived by applying the conservation law on multiple control volumes and assuming a two wave model. The HLL solver and its improved HLLC (adding a third wave to better resolve contact discontinuities) are still widely used in numerical methods for the Euler equations of gas dynamics for their simplicity and decent accuracy.

Let's consider a Riemann problem with left and right states $(\mathbf{q}_L, \mathbf{q}_R)$. The HLL solver requires approximate value or the smallest and largest wave speeds S_L and S_R. We assume a two wave configuration separating three constant states $\mathbf{q}_L, \mathbf{q}_{\text{HLL}}, \mathbf{q}_R$ as represented in (19). We assume that the timestep Δt and cell size $(x_R - x_L)$ is chosen such that $x_L \le \Delta t S_L$ and $\Delta t S_R \le x_R$. The conservation law integrated over the control volume $[x_L, x_R] \times [0, \Delta t]$ yields:

$$\begin{align} \int_{x_L}^{x_R} \mathbf{q}(x, \Delta t)\,dx &= \int_{x_L}^{x_R} \mathbf{q}(0, x)\,dx + \int_0^{\Delta t} \mathbf{f}(\mathbf{q}(x_L, t))\,dt - \int_0^{\Delta t} \mathbf{f}(\mathbf{q}(x_R, t))\,dt \tag{3} \\ &= x_R \mathbf{q}_R - x_L \mathbf{q}_L + \Delta t(\mathbf{f}_L - \mathbf{f}_R), \tag{4} \end{align}$$

where $\mathbf{f}_L = \mathbf{f}(\mathbf{q}_L)$ and $\mathbf{f}_R = \mathbf{f}(\mathbf{q}_R)$. Splitting the left hand side of (3) into three terms:

$$\int_{x_L}^{x_R} \mathbf{q}(\Delta t, x)\,dx = \int_{x_L}^{\Delta t S_L} \mathbf{q}(\Delta t, x)\,dx + \int_{\Delta t S_L}^{\Delta t S_R} \mathbf{q}(\Delta t, x)\,dx + \int_{\Delta t S_R}^{x_R} \mathbf{q}(\Delta t, x)\,dx,$$

and evaluating the first and third integrals by applying the integral form of the conservation law on the control volumes $[x_L, S_L\Delta t] \times [0, \Delta t]$ and $[S_R\Delta t, x_R] \times [0, \Delta t]$ yields:

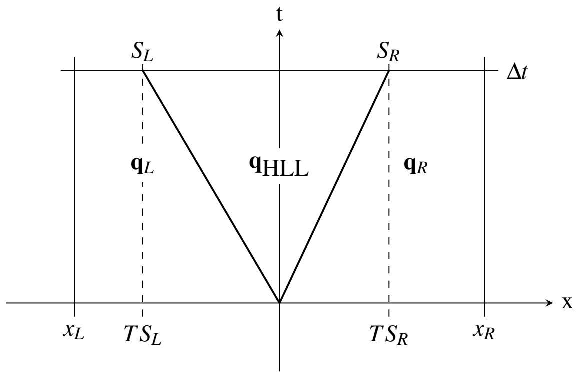


Figure 19: wave structure assumed for the HLL solver.

$$\int_{x_L}^{x_R} \mathbf{q}(\Delta t, x)\, dx = \int_{\Delta t S_L}^{\Delta t S_R} \mathbf{q}(\Delta t, x)\, dx + (\Delta t S_L - x_L)\mathbf{q}_L + (x_R - \Delta t S_R)\mathbf{q}_R. \tag{5}$$

Combining (3) and (5) produces:

$$\int_{\Delta t S_L}^{\Delta t S_R} \mathbf{q}(\Delta t, x)\, dx = \Delta t \left(S_R \mathbf{q}_R - S_L \mathbf{q}_L + \mathbf{f}_L - \mathbf{f}_R \right).$$

By dividing this by $\Delta t(S_R - S_L)$, we get the average state in-between the two waves at $t = \Delta t$ that we set to be the intermediate state $\mathbf{q}_{\text{HLL}}$:

$$\mathbf{q}_{\text{HLL}} := \frac{1}{\Delta t(S_R - S_L)} \int_{\Delta t S_L}^{\Delta t S_R} \mathbf{q}(\Delta t, x)\, dx = \frac{S_R \mathbf{q}_R - S_L \mathbf{q}_L + \mathbf{f}_L - \mathbf{f}_R}{S_R - S_L}. \tag{6}$$

We now make the further assumption that $x_L \leq 0$ and $x_R \geq 0$. By evaluating the integral form of the conservation law on the control volume $[x_L, 0] \times [0, \Delta t]$ or and substituting $\mathbf{q}_{\text{HLL}}$ by (6), we get:

$$\begin{aligned} \mathbf{f}_{0,L} &= \mathbf{f}_L + S_L(\mathbf{q}_{\text{HLL}} - \mathbf{q}_L) \\ &= \frac{S_R \mathbf{f}_L - S_L \mathbf{f}_R + S_L S_R \left(\mathbf{q}_R - \mathbf{q}_L \right)}{S_R - S_L}. \end{aligned}$$

We notice that doing the same computation on the control volume $[0, x_R] \times [0, \Delta t]$ yields the consistency condition $\mathbf{f}_{0,L} = \mathbf{f}_{0,R}$. The HLL flux is then defined as:

$$\mathbf{f}_{\text{HLL}} := \begin{cases} \mathbf{f}_L, & \text{if } 0 \leq x_L, \\ \dfrac{S_R \mathbf{f}_L - S_L \mathbf{f}_R + S_L S_R \left(\mathbf{q}_R - \mathbf{q}_L \right)}{S_R - S_L} & \text{if } S_L \leq 0 \leq S_R, \\ \mathbf{f}_R & \text{if } 0 \geq S_R. \end{cases}$$

The KEPES flux is another numerical flux for the compressible Euler equation introduced by Praveen Chandrashekar in [4]. We have chosen to use the KEPES flux as well as it is more computationally expensive than the rather simple HLL flux and is thus better suited for GPU computation. Therefore, we have decided to implement both HLL and KEPES fluxes to observe if such numerical flux better utilize GPU resources.

4.4 Validation

In order to validate the implementation of the Godunov scheme, after verifying basic properties of the scheme (testing conservation for example and trivial examples), we have used the following two classical test cases in the following subsections.

4.4.1 Sod shock tube

The Sod shock tube is a common test case for numerical schemes for solving the compressible Euler equations for an ideal gas. It is a Riemann problem with left and right states:

$$\begin{bmatrix}\rho_L\\P_L\\u_L\end{bmatrix}=\begin{bmatrix}1.0\\1.0\\0.0\end{bmatrix},\quad\begin{bmatrix}\rho_R\\P_R\\u_R\end{bmatrix}=\begin{bmatrix}0.125\\0.1\\0.0\end{bmatrix}.$$

This test case contains the tree kind of discontinuities encoutered for the Euler equations, namely a rarefaction wave, contact discontinuity and a shock discontinuity. We notice in figure (20), that both HLL and KEPES fluxes together with the Godunov scheme correctly resolve the expansion fan, contact discontinuity and shock discontinuity. However, a great deal of numerical diffusion is present around the contact discontinuity. For the HLL flux, this is easily explained by the fact that the simple two wave model that doesn't take into account the contact discontinuity.

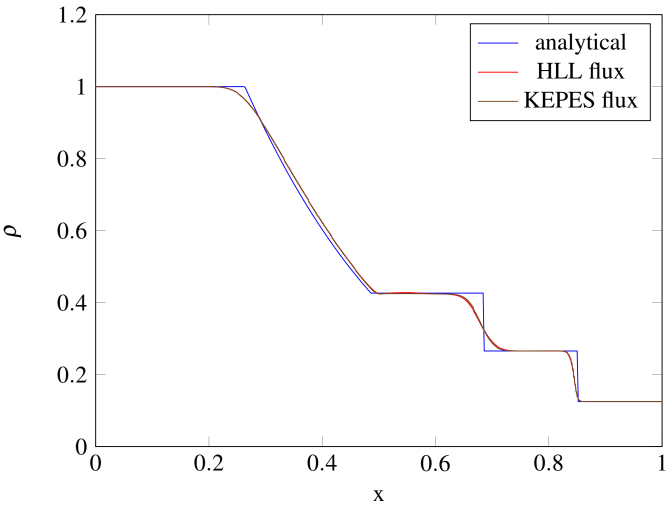


Figure 20: density profile at $t = 0.2$.

4.4.2 Isentropic Euler vortex problem

The vortex in isentropic flow [17] is one of the few exact non-trivial solutions for the compressible Euler equations in 2D that can be used to numerically compute the convergence rate of a scheme as the analytical solution is smooth. This test case consists of a convection of an isentropic vortex in inviscid flow. It can be constructed by the superposition of the uniform free-stream conditions $\rho = 1, \mathbf{u} = \mathbf{u}_\infty \in \mathbb{R}^2, p = 1$ perturbed by the velocity field:

$$\begin{bmatrix}\delta u\\\delta v\end{bmatrix}=\frac{\beta}{2\pi}\exp\left(\frac{1-r^2}{2}\right)\begin{bmatrix}-(y-y_o)\\(x-x_o)\end{bmatrix},$$

with $\beta \in \mathbb{R}$, $(x_0, y_0) \in \mathbb{R}^2$ and $r = \sqrt{(x-x_0)^2+(y-y_0)^2}$. We then get the initial condition:

$$\begin{cases}\rho &= \left[1-\dfrac{(\gamma-1)\beta^2}{8\gamma\pi}e^{1-r^2}\right]^{\frac{1}{\gamma-1}},\\ \rho u &= \rho(u_\infty+\delta u)=\rho\left[1-\dfrac{\beta}{2\pi}e^{\frac{1-r^2}{2}}\right],\\ \rho v &= \rho(v_\infty+\delta v)=\rho\left[1+\dfrac{\beta}{2\pi}e^{\frac{1-r^2}{2}}\right],\\ E &= \dfrac{p}{\gamma-1}+\dfrac{1}{2}\rho(u^2+v^2).\end{cases}$$

We chose the domain $[-5,5]^2$ with periodic boundary conditions, $x_0 = y_0 = 0$ and $\beta = 5, u_\infty = 1, v_\infty = 1$ (so that the vortex flow is negligible near the boundary of the domain closely resembling the free-stream condition chosen). As the isentropic flow with null free-stream condition is stationary, we expect with the chosen $\mathbf{u}_\infty$ the flow to be periodic with period 10. The convergence results for the Godunov scheme using both HLL and KEPES fluxes are given in figure 21. The absolute errors are computed after one period. As expected, we observe an order one convergence rate for the scheme with both numerical fluxes.

	KEPES		HLL	
h	error	order	error	order
2^{-4}	1.63×10^{-3}	–	2.22×10^{-3}	–
2^{-5}	1.38×10^{-3}	0.24	1.63×10^{-3}	0.44
2^{-6}	9.39×10^{-4}	0.56	1.01×10^{-3}	0.68
2^{-7}	5.77×10^{-4}	0.78	5.67×10^{-4}	0.83
2^{-8}	2.95×10^{-4}	0.89	3.00×10^{-4}	0.92
2^{-9}	1.53×10^{-4}	0.95	1.55×10^{-4}	0.96

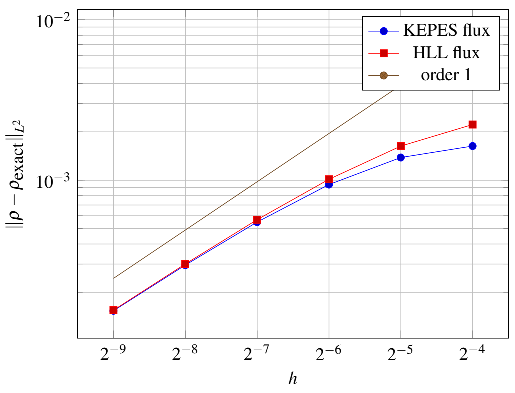


Figure 21: convergence of the Godunov scheme.

5 Results and performance analysis

5.1 Example showcase

To show the capabilities of the t8gpu library, I have run multiple simulations in 2D and 3D. Here are described some of them:

- The figure 22 depicts a simulation ran using t8gpu using subgrid elements and a gradient based refinement criteria in 2D together with the refinement level. The initial conditions inspired by [15] are:

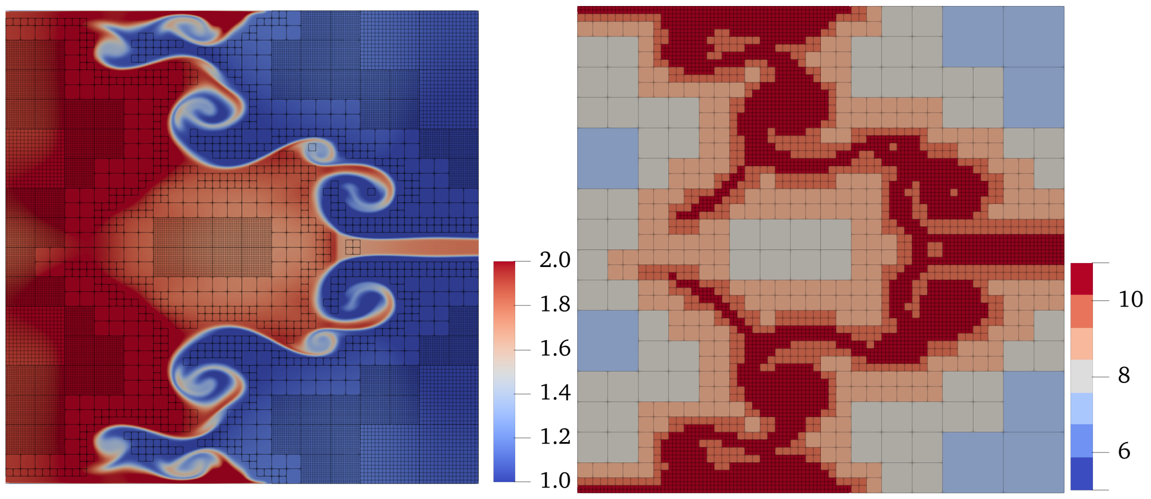


Figure 22: Kelvin-Helmholtz instability with reflective boundary conditions using 16×16 subgrid elements. The left plot represents the density field for $t = 1.0$. The grid and subgrid are represented. For legibility reason, the smaller subgrid and grid are not represented. The plot on the right represents the refinement level.

$$p = 2.5,$$
$$\rho(x,y) = \begin{cases} 2 & \text{if } |x| \leq 0.25, \\ 1 & \text{otherwise,} \end{cases}$$
$$v_x(x,y) = \begin{cases} 0.5 & \text{if } |x| \leq 0.25, \\ -0.5 & \text{otherwise,} \end{cases}$$
$$v_y(x,y) = w_0 \sin(4\pi x) \left[\exp\left(-\frac{(y-0.25)^2}{2\sigma^2} \right) + \exp\left(-\frac{(y+0.25)^2}{2\sigma^2} \right) \right] \text{sign}(y),$$

with $\sigma = 0.035$.

The domain is $\Omega = [-0.5, 0.5]^2$ with reflective boundary conditions:

$$\forall \mathbf{x}_0 \in \partial\Omega, \begin{cases} \rho(\mathbf{x_0}) & = \lim\limits_{\substack{\mathbf{x}\to\mathbf{x_0} \\ \mathbf{x}\neq\mathbf{x}_0}} \rho(x), \quad E(\mathbf{x_0}) = \lim\limits_{\substack{\mathbf{x}\to\mathbf{x_0} \\ \mathbf{x}\neq\mathbf{x}_0}} E(x), \\ u_\perp(\mathbf{x_0}) & = -[\lim\limits_{\substack{\mathbf{x}\to\mathbf{x_0} \\ \mathbf{x}\neq\mathbf{x}_0}} u_\perp(x)], \\ u_{||}(\mathbf{x_0}) & = \lim\limits_{\substack{\mathbf{x}\to\mathbf{x_0} \\ \mathbf{x}\neq\mathbf{x}_0}} u_{||}(x). \end{cases}$$

With this initial condition and the refinement criteria described in a previous section, we notice that the mesh refinement closely follows the boundary between the low density and high density fluids from the initial condition. Moreover, the symmetry of the test case with respec t to the x-axis is kept throughout the simulation as expected.

- The figure 23 is the exact same test case as in [15] using different subgrid sizes (16×16 and 32×32) but the same total number of sugrid mesh elements to test the implementation of the numerical scheme. As expected, the finaly density distribution is identical with both subgrid sizes.

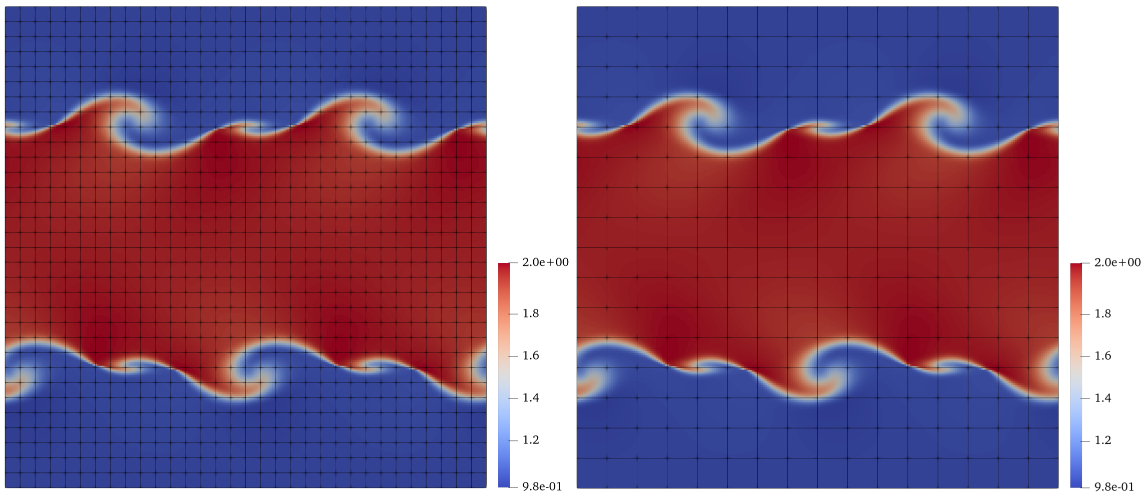


Figure 23: Kelvin-Helmholtz instability using a 32×32 grid of 8×8 subgrid elements on the left and 16×16 grid of 16×16 subgrid elements without mesh refinement. We notice that, as expected, both simulation yield the same identical solution.

- Figure 24 represents the density of a 2D Riemann problem introduced in [14]. This is a generalization of Riemann problems to 2D. The plane is divided in 4 quadrants with different initial conditions:

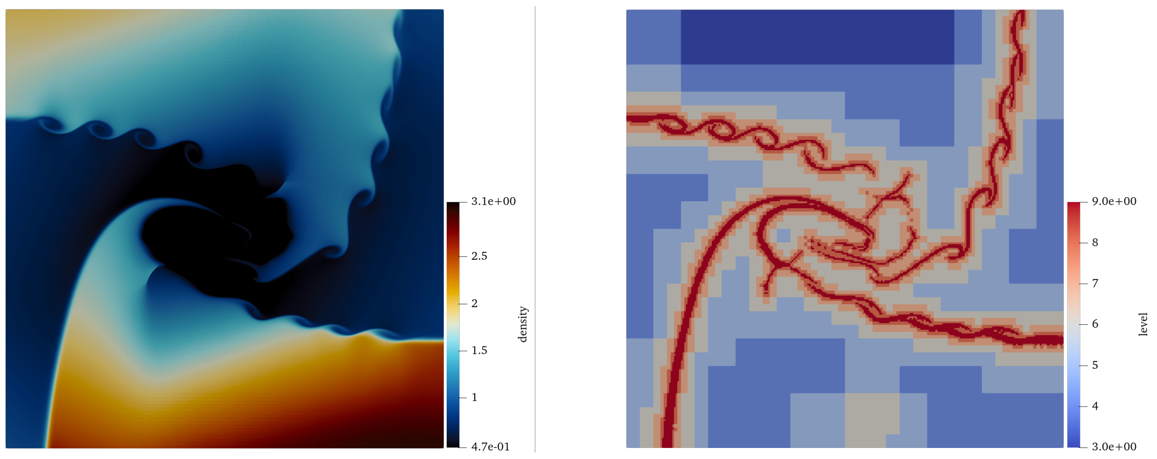
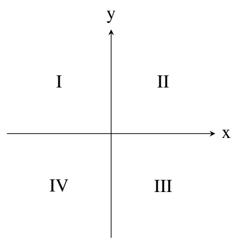


Figure 24: density field and refinement level of 2D Riemann problem.



$$\begin{cases} (\rho_{\mathrm{I}}, U_{\mathrm{I}}, V_{\mathrm{I}}, p_{\mathrm{I}}) & = (1.0, -0.75, -0.5, 1.0), \\ (\rho_{\mathrm{II}}, U_{\mathrm{II}}, V_{\mathrm{II}}, p_{\mathrm{II}}) & = (2.0, -0.75, \quad 0.5, 1.0), \\ (\rho_{\mathrm{III}}, U_{\mathrm{III}}, V_{\mathrm{I}}, p_{\mathrm{III}}) & = (1.0, \quad 0.75, \quad 0.5, 1.0), \\ (\rho_{\mathrm{IV}}, U_{\mathrm{IV}}, V_{\mathrm{IV}}, p_{\mathrm{IV}}) & = (3.0, \quad 0.75, -0.5, 1.0). \end{cases}$$

- Figure 25 represents the density field obtained simulation the Kelvin-Helmholtz instability on a 3D spherical-shell. This demonstrates the ability of t8gpu to treat AMR with meshes containing prism elements as well as geometry-aware AMR (by this I mean the ability to refine elements to closer match a certain geometry: in this case a 3D shell) as thanks to t8code's support of those features. The domain is $\Omega = \{\mathbf{x} \in \mathbb{R}^3 | r_{\text{min}} \leq |\mathbf{x}| \leq r_{\text{max}}\}$ with $r_{\text{min}} = 0.8, r_{\text{max}} = 1.0$. The initial conditions in spherical coordinates are:

$$\rho(r,\varphi,\theta) = \begin{cases} 2 & \text{if } \theta \leq \pi \\ 1 & \text{otherwise} \end{cases}, \quad p(r,\varphi,\theta) = 2.5,$$

$$u_\varphi(r,\varphi,\theta) = \begin{cases} \frac{1}{2} r \sin(\theta) & \text{if } \theta \leq \pi, \\ -\frac{1}{2} r \sin(\theta) & \text{otherwise,} \end{cases}$$

$$v_\theta(r,\varphi,\theta) = \frac{1}{2} r \sin(2\varphi) \exp\left(\left[-\frac{(\theta-\pi)}{2\sigma}\right]^2\right), \quad u_r(r,\varphi,\theta) = 0,$$

with $\sigma = 0.15, t_{\text{final}} = 1.0$ and reflective boundary conditions.

5.2 Motivating example of AMR with subgrid elements

As a example of the possible gain that we can obtain using AMR, I ran the validation case of section 4.4.2 using subgrid elements with and without AMR. Using a uniform grid, we get an absolute error of $5.1 \cdot 10^{-4}$ and the simulation ran in 19.4 seconds using 9 MPI ranks. With AMR, we get the absolute error $5.6 \cdot 10^{-4}$ and the simulation ran in 2.7 seconds with the same number of MPI ranks. Thus, for a similar error, we get a speedup of $\times 7$.

5.3 Performance analysis

All performance results were obtained on a compute node with two Intel Xeon Gold 6258R CPUs and one Nvidia RTX A6000 GPU. It important to note that the RTX A6000 single precision floating point performance

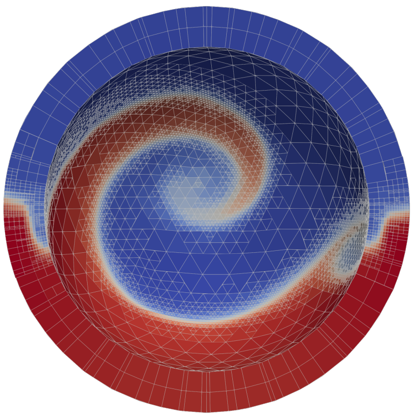


Figure 25: Kelvin-Helmholtz instability in a 3D spherical shell using AMR on a mesh of prism elements.

to double precision ratio is a staggering 64. Even though scientific computing usually requires the full 64 bit double precision range, we will analyze performance in single and double precision. A lot of modern GPU for HPC nowadays have the same performance in single precision and double precision.

5.3.1 First approach

Kernel performance The naive approach uses two kernels:

- The most important kernel is the flux computation kernel using either the HLL or KEPES flux. We notice in the roofline model 26 that both flux computation kernels in double precision are compute bound as their arithmetic intensity is high enough to reach the compute bound region. Even though the KEPES flux kernel has a higher arithmetic intensity, this kernel performs worse than the HLL flux computation kernel. As for these kernels in single precision, this time we are memory bound. As the KEPES flux kernel has double the arithmetic intensity as the HLL flux kernel, this kernel is able to reach 6 TFLOPS while the HLL kernel only reaches 3.5 TFLOPS. Here, the vastly different performance profile of this GPU in single and double precision makes a significant difference when it comes to the better performing flux computation kernel. Indeed, in double precision, there does not seem to be any performance benefit in using the KEPES flux rather than the HLL flux (other than for accuracy considerations). However, in single precision, the more computationally expensive KEPES flux allows us to more efficiently use the GPU resources.
- The second important GPU kernel is the timestepping kernel (there are actually three similar kernels to handle the 3 stages of the Runge Kutta integration scheme chosen that have identical performance). For these kernels, the arithmetic intensity is so low that the performance in single precision and double precision is almost identical and we only reach 80 GFLOPS. In this approach, there is no way around it as we cannot merge the flux computation kernel with the timestepping kernel as they iterate on different entities: the flux computation kernel iterates on the faces and timestepping kernel on the mesh elements. The subgrid approach will be able to fuse the flux computation and timestepping kernels yielding better GPU utilization.

Whole program performance Figure 28 depicts a timeline graph of the simulation ran for the last showcase example. Even though 50 solver steps are executed in-between the mesh adaptation routines, we notice that the majority of the runtime is spent on the CPU side doing mesh related operation. Thus, as expected AMR using this approach is detrimental as the mesh adaptation routines on the CPU are slow compared to the solver iteration on the GPU. This is due to the highly parallel architecture of the GPU. Moreover, the mesh adaptation routines on the CPU require much more process to process communication compared to the almost embarassingly parallel nature of the numerical solver. This is why the subgrid approach is considered in the following section.

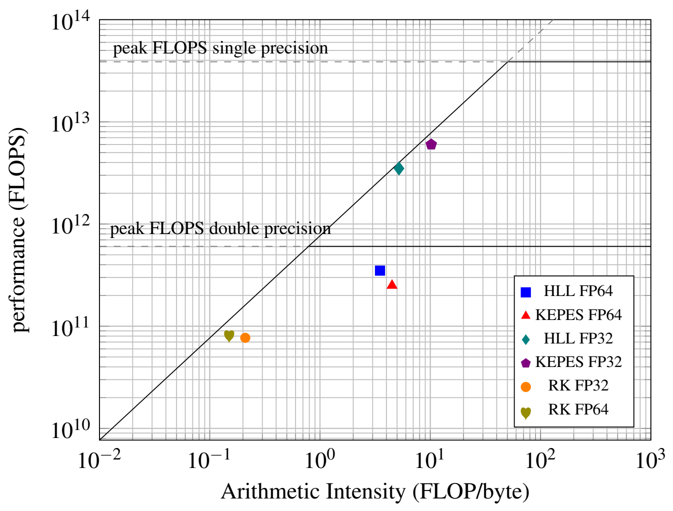


Figure 26: roofline model for the flux computation and timestepping kernels in single and double precision.

5.3.2 Second approach: AMR with subgrid elements

Starting from this section, we will only consider the KEPES flux.

Kernel performance The subgrid approach uses three types of kernels:

- A first kernel is necessary to compute the inner fluxes. This kernel uses a GPU grid composed of blocks (or CTA) the size of the subgrid (here we have chosen 16×16 subgrid in 2D). The number of blocks is the number of global mesh elements. We expect here the cartesian arrangement of the subgrid elements within a block to yield better GPU utilization as all global memory accesses are now coalesced and we can make good use here of local memory to store intermediary results efficiently. Moreover, we expect the arithmetic intensity to be higher because in 2D, for the same amount of bytes fetched per GPU thread, we are able to compute two fluxes instead of one (we compute fluxes in both x and y direction whereas the naive approach computes one flux per thread per face). However, this higher arithmetic intensity does not seem to greatly increase performance compared to the naive approach as we get in single precision 6.5 TFLOPS similarily to the compute kernel of the naive approach. In double precision, as we were already compute bound in the naive approach, this does not yield any more performance.
- Another kernel is needed to compute the remaining fluxes in-between mesh elements. The grid for this kernel iterates over face elements. As seen in figure 27, this kernel's performance is poor compared to the inner flux kernel. This is due to a very low arithmetic intensity and possibly wrong choice of GPU kernel grid size.
- As it currently stands, additional kernels for the timestepping are used that exhibit similar performance to the ones in figure 26. However, those kernel could be fused into the inner flux computation kernel. That would get rid of the low performing timestepping kernels and reduce the overhead of GPU kernel launch from the CPU.

Whole program performance Figure 30 depicts a timeline graph of the simulation ran for the first showcase example. This time, we observe that mesh adaptation routines take up a minor fraction of the runtime. Moreover, GPU utilization is higher.

Figure 29 focuses on a single solver iteration. We notice that the GPU utilization is rather low. This is expected as each ranks possesses its own CUDA context and a single CUDA context can run at the same time on the GPU by default. Thus, as kernels are launched by all MPI ranks, the GPU needs to do multiple context switches which are really expensive. Fortunately, the Multi-Process Service (MPS) is an alternative implementation of the CUDA runtime provided by Nvidia that was designed for programs using MPI together with CUDA. It allows multiple CPU processes to share the same CUDA context and thus avoid expensive

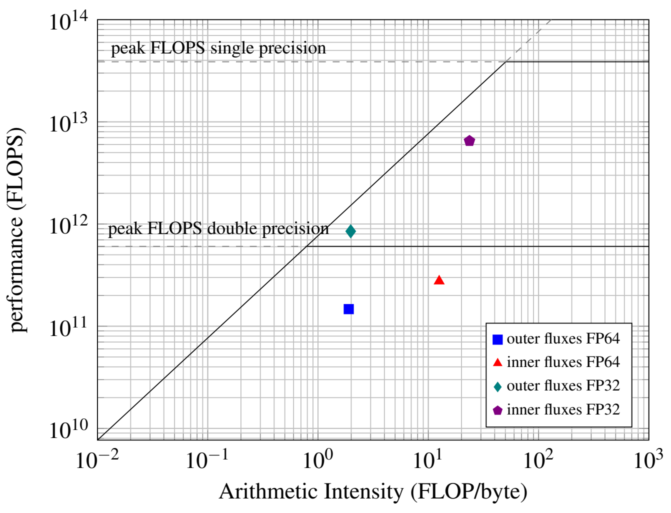


Figure 27: roofline model for the inner and outer flux computation kernels in single and double precision using subgrid elements.

context switches. When activating the MPS, the GPU utilization looks like figure 31. Here, we notice that kernels launched from different processes are able to overlap and we get a way higher GPU utilization. We also notice that the runtime taken by the timestepping kernels is substantial. Fusing those kernels with inner flux computation would results in a big efficiency gain.

Moreover, on the CPU side, we notice that the majority of the runtime is spent on synchronization primitives rather than doing any meaningful work. One way to better use the CPU during solver iterations would be to do the mesh adaptation asynchronously on a copy of the mesh while doing solver iterations on the GPU. This would also hide most of the overhead of the AMR routines.

6 Conclusion and perspectives

To conclude, I have sucessfully designed and implemented a finite volume library using the dynamic AMR library t8code to target GPUs. This framework uses modern C++17 and CUDA to be able to target GPUs as their use in HPC is becoming prevalent. To test this library, I have implemented a finite volume solver for the compressible Euler equations in 2D/3D using a Godunov scheme and the HLL and KEPES fluxes. Furthermore, a naive first approach was implemented that was shown to have poor performance. Then, a subgrid approach was shown to yield way better performance and be a proof of concept for further developments. From what I have developed, I have identified the following possible improvements:

- Multi-GPU support: for now t8gpu only uses one GPU. To be able to target supercomputers with thousand of nodes and multiple GPUs per nodes, some data structures needs to be rethought.
- higher order scheme: a higher order scheme would better represent modern numerical schemes (using WENO reconstruction for instance or a discontinuous galerkin method).
- support for multiple element types with subgrids: one of the strenghs of t8code is its ability to use mesh with multiple mesh elements types (hexahedron, prism, tetrahedron) all within the same mesh. Thus, extending subgrid elements to those elements types is important to support one of t8code's most important feature.

A Appendix

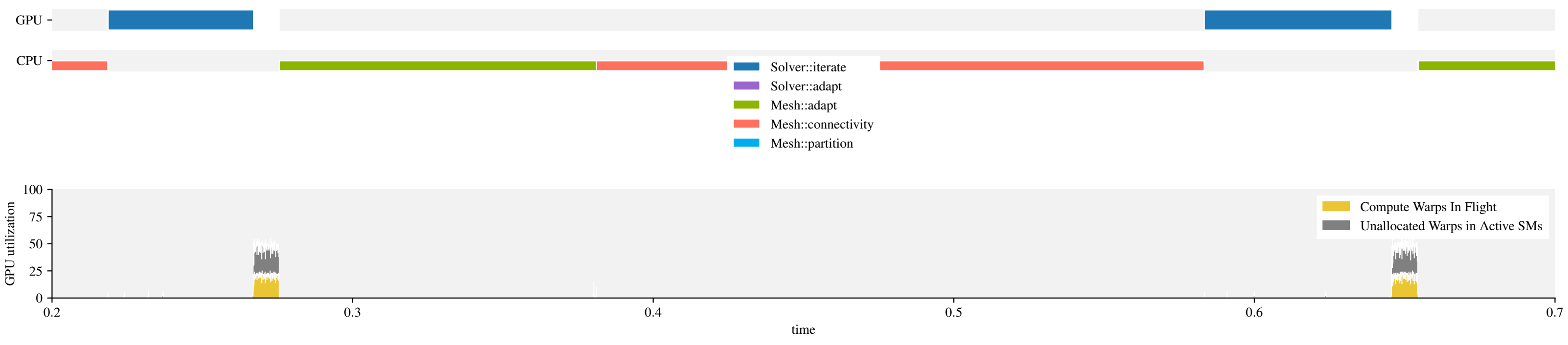


Figure 28: timeline graph of a numerical simulation using the first approach.

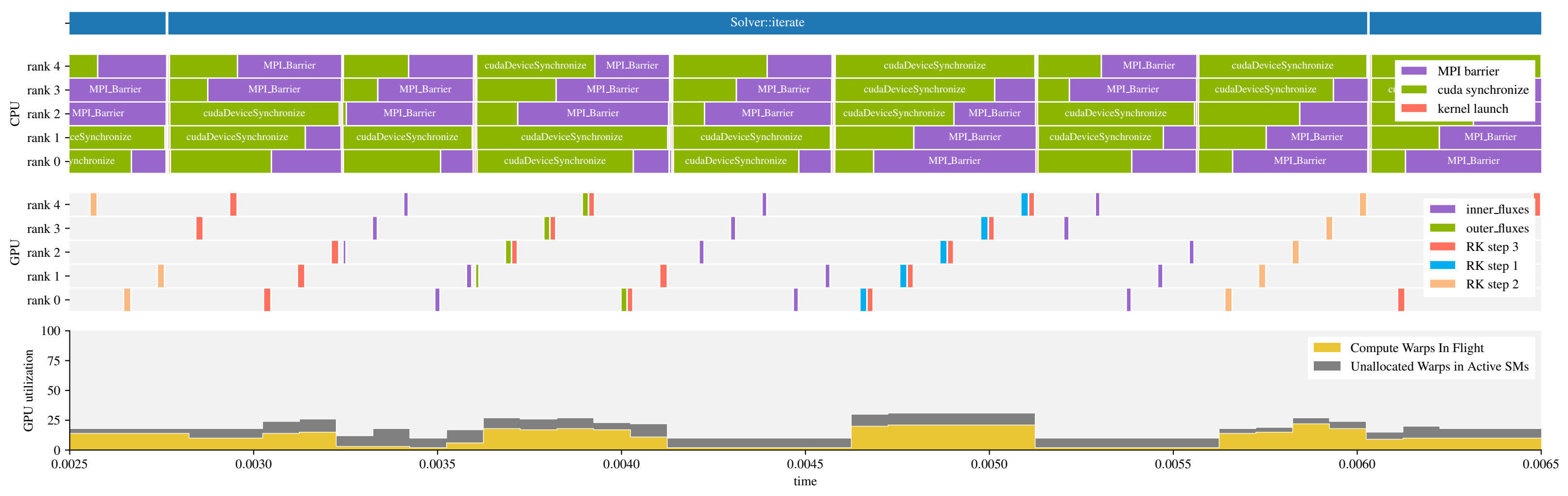


Figure 29: timeline graph of a solver iteration without using the MPS.

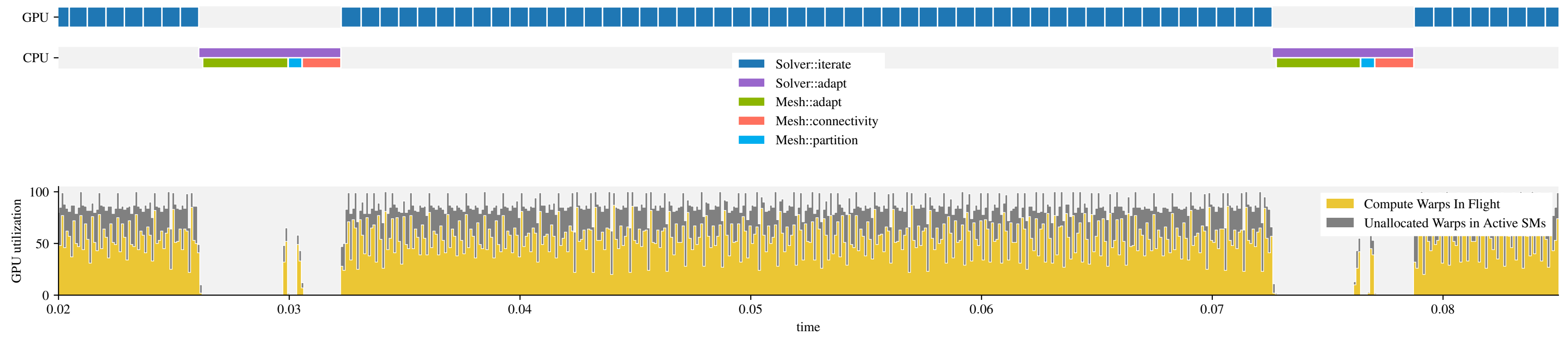


Figure 30: timeline graph of solver iterations and mesh adaptation routines.

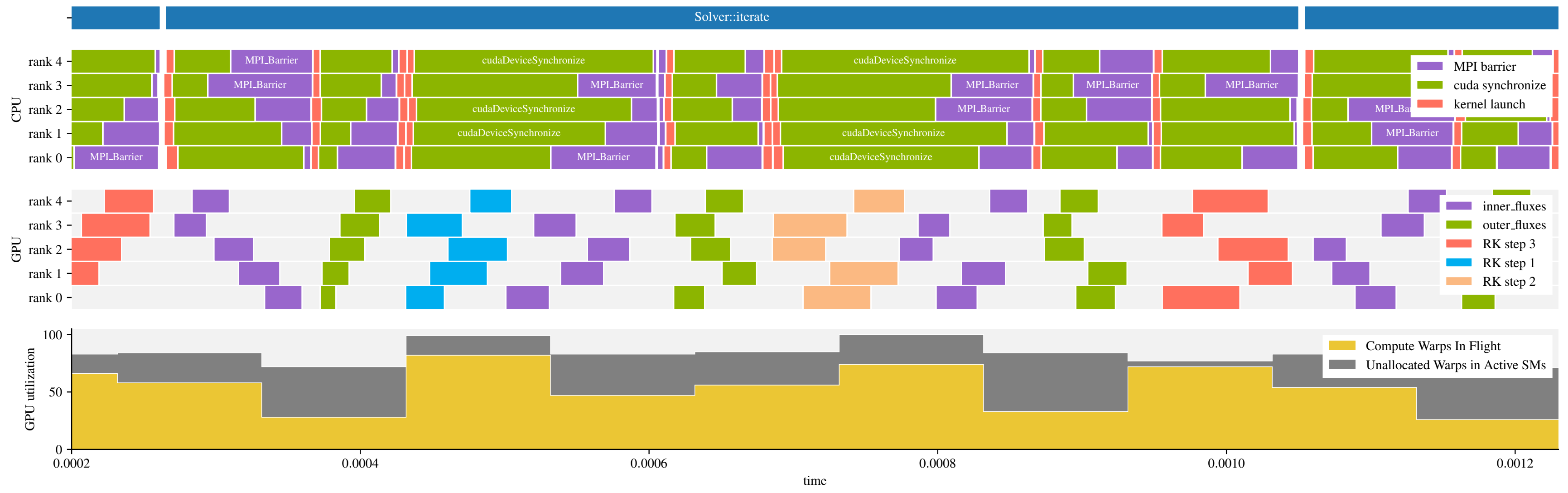


Figure 31: timeline graph of a solver iteration using subgrid elements.

B Glossary

AMR	adaptive mesh refinement
API	application programming interface
AoS	array of structures
CPU	central processing unit
CUDA	compute unified device architecture
device	designates a GPU
FLOPS	floating point operations per second
GPGPU	general purpose graphics processing unit
GPU	graphics processing unit
host	designates a CPU responsible for queuing up GPU kernels
IPC	interprocess communication
MPI	message passing interface
MPS	multi-process service
PCIe	Peripheral Component Interconnect Express
SIMD	single instruction, multiple data
SIMT	single instruction, multiple threads
SM	streaming multiprocessor
SoA	structure of arrays
SPMD	single program, multiple data

References

[1] David A. Beckingsale et al. *Parallel block structured adaptive mesh refinement on graphics processing units*. 2014. url: `https://wrap.warwick.ac.uk/id/eprint/90866/`.

[2] Carsten Burstedde and Johannes Holke. "A Tetrahedral Space-Filling Curve for Nonconforming Adaptive Meshes". In: *SIAM Journal on Scientific Computing* 38.5 (2016), pp. C471–C503. doi: `10.1137/15M1040049`. eprint: `https://doi.org/10.1137/15M1040049`. url: `https://doi.org/10.1137/15M1040049`.

[3] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. "p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees". In: *SIAM Journal on Scientific Computing* 33.3 (2011), pp. 1103–1133. doi: `10.1137/100791634`.

[4] Praveen Chandrashekar. "Kinetic Energy Preserving and Entropy Stable Finite Volume Schemes for Compressible Euler and Navier-Stokes Equations". In: *Communications in Computational Physics* 14.5 (Nov. 2013), pp. 1252–1286. issn: 1991-7120. doi: `10.4208/cicp.170712.010313a`. url: `http://dx.doi.org/10.4208/cicp.170712.010313a`.

[5] Andrew Giuliani and Lilia Krivodonova. "Adaptive mesh refinement on graphics processing units for applications in gas dynamics". In: *Journal of Computational Physics* 381 (2019), pp. 67–90. issn: 0021-9991. doi: `https://doi.org/10.1016/j.jcp.2018.12.019`. url: `https://www.sciencedirect.com/science/article/pii/S0021999118308155`.

[6] Sigal Gottlieb, Chi-Wang Shu, and Eitan Tadmor. "Strong Stability-Preserving High-Order Time Discretization Methods". In: *SIAM Review* 43.1 (2001), pp. 89–112. doi: `10.1137/S003614450036757X`. eprint: `https://doi.org/10.1137/S003614450036757X`. url: `https://doi.org/10.1137/S003614450036757X`.

[7] Amiram Harten, Peter D. Lax, and Bram van Leer. "On Upstream Differencing and Godunov-Type Schemes for Hyperbolic Conservation Laws". In: *SIAM Review* 25.1 (1983), pp. 35–61. doi: `10.1137/1025002`. eprint: `https://doi.org/10.1137/1025002`. url: `https://doi.org/10.1137/1025002`.

[8] Johannes Holke. *Scalable Algorithms for Parallel Tree-based Adaptive Mesh Refinement with General Element Types*. 2018. arXiv: `1803.04970 [cs.DC]`. url: `https://arxiv.org/abs/1803.04970`.

[9] Johannes Holke, David Knapp, and Carsten Burstedde. *An Optimized, Parallel Computation of the Ghost Layer for Adaptive Hybrid Forest Meshes*. 2019. arXiv: `1910.10641 [cs.DC]`. url: `https://arxiv.org/abs/1910.10641`.

[10] Johannes Holke et al. *t8code*. Version v2.0.0. Apr. 2024. doi: `10.5281/zenodo.10996663`. url: `https://doi.org/10.5281/zenodo.10996663`.

[11] Maël Karembe, Johannes Markert, and Johannes Holke. *t8gpu*. Oct. 2024. url: `https://github.com/DLR-AMR/t8gpu`.

[12] Randall J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2002.

[13] I Menshov and P Pavlukhin. "GPU-native gas dynamic solver on octree-based AMR grids". In: *Journal of Physics: Conference Series* 1640.1 (2020), p. 012017. doi: `10.1088/1742-6596/1640/1/012017`. url: `https://dx.doi.org/10.1088/1742-6596/1640/1/012017`.

[14] Liang Pan, Jiequan Li, and Kun Xu. *A Few Benchmark Test Cases for Higher-order Euler Solvers*. 2016. arXiv: `1609.04491 [math.NA]`. url: `https://arxiv.org/abs/1609.04491`.

[15] Kevin Schaal et al. "Astrophysical hydrodynamics with a high-order discontinuous Galerkin scheme and adaptive mesh refinement". In: *Monthly Notices of the Royal Astronomical Society* 453.4 (Sept. 2015), pp. 4279–4301. issn: 1365-2966. doi: `10.1093/mnras/stv1859`. url: `http://dx.doi.org/10.1093/mnras/stv1859`.

[16] Hsi-Yu Schive, Yu-Chih Tsai, and Tzihong Chiueh. "GAMER □: A GRAPHIC PROCESSING UNIT ACCELERATED ADAPTIVE-MESH-REFINEMENT CODE FOR ASTROPHYSICS". In: *The Astrophysical Journal Supplement Series* 186.2 (Feb. 2010), pp. 457–484. issn: 1538-4365. doi: `10.1088/0067-0049/186/2/457`. url: `http://dx.doi.org/10.1088/0067-0049/186/2/457`.

[17] H.C Yee, N.D Sandham, and M.J Djomehri. "Low-Dissipative High-Order Shock-Capturing Methods Using Characteristic-Based Filters". In: *Journal of Computational Physics* 150.1 (1999), pp. 199–238. issn: 0021-9991. doi: `https://doi.org/10.1006/jcph.1998.6177`. url: `https://www.sciencedirect.com/science/article/pii/S0021999198961770`.

[18] Ui-Han Zhang, Hsi-Yu Schive, and Tzihong Chiueh. "Magnetohydrodynamics with GAMER". In: *The Astrophysical Journal Supplement Series* 236.2 (June 2018), p. 50. issn: 1538-4365. doi: `10.3847/1538-4365/aac49e`. url: `http://dx.doi.org/10.3847/1538-4365/aac49e`.

[19] Weiqun Zhang et al. "AMReX: a framework for block-structured adaptive mesh refinement". In: *Journal of Open Source Software* 4.37 (May 2019), p. 1370. doi: `10.21105/joss.01370`. url: `https://doi.org/10.21105/joss.01370`.